Multi-resource management in embedded real-time systems

Mike Holenderski
## Contents

1 System model ................................................................. 3  
   1.1 Resource model ......................................................... 3  
   1.2 Application model .................................................. 5  
   1.3 Mapping ............................................................... 10  
   1.4 Platform ............................................................... 12  

2 Multi-resource management ............................................... 13  
   2.1 Problem description ................................................ 14  
   2.2 Related work ......................................................... 14  
   2.3 Towards multi-resource sharing ................................... 21  
   2.4 Parallel-SRP (PSRP) .................................................. 23
Chapter 1

System model

A system consists of applications, resources and a mapping between them. Application workload is expressed in terms of tasks, where a task represents the work which needs to be done in response to an event, such as processing a newly arrived video frame. The mapping is the main focus of this thesis. It describes how the “ownership” of resources changes during runtime, i.e. which task “owns” a particular resource at a particular time. The ownership may change, e.g. if several tasks need to access the same memory region or the same processor. A mapping must satisfy certain constraints. In real-time systems the constraints are expressed in terms of task deadlines. Embedded systems exhibit additional constraints which address the overheads associated with the mapping due to resource constraints.

Notation  We will use the dot notation to refer to individual static parameters, e.g. $\tau_i.E$ for the worst-case execution time of task $\tau_i$, and function notation for dynamic parameters which change during runtime, e.g. $\beta(t, b)$ for the remaining capacity of budget $b$ at time $t$.

1.1 Resource model

The basic blocks of modern computers are transistors. They can be regarded as the fundamental resources provided to the application tasks. In the end our goal is to map the transistors to tasks and to describe how this mapping changes during runtime. Since modern computers contain billions of transistors, it is not feasible to reason about individual transistors. It has therefore become a custom to group transistors according to the function they perform, e.g. a processor or a bus, performing a certain function is referred to as a physical resource.

The purpose of a resource model is to provide a certain level of abstraction helping to describe the mapping of physical resources to tasks. It has to be simple enough to reason about, while at the same time expressive enough to use the available resources efficiently. At the core of our resource model is the multi-unit resource.
1.1.1 Multi-unit resources

Let $I$ be a set of identifiers, and let $R \subseteq I \times \mathbb{N}$ be the set of all multi-unit resources in the system.

**Definition 1.1 (Multi-unit resource).** A multi-unit resource $r \in R$ is a tuple $(id, N)$, where $r.id$ is a resource identifier and $r.N$ is its capacity. The capacity represents the maximum number of units the resource can provide simultaneously.

Memory is an example of a multi-unit resource. In this thesis, when talking about the memory resource we are interested in the memory requirements in terms of memory space, and ignore the specifics of memory allocation and the actual data stored in the memory. A memory managed as a collection of fixed-sized blocks, can be regarded as a multi-unit resource with capacity equal to the number of blocks. In this sense our multi-unit resource is similar to a multi-unit resource discussed by Baker (1991).

The capacity of a multi-unit resource represents essentially the number of tasks which can use the resource simultaneously. A multi-core processor can therefore be modeled as a resource with capacity equal to the number of cores.

**Notation**

Let $A, B \subseteq R$. For ease of presentation, we will sometimes write $r \in A$ to designate $r \in R \land (\exists k \in A : k.id = r.id)$. Similarly, we will write $A \cap B$ to designate $\{r|r \in R \land (\exists k \in A : k.id = r.id) \land (\exists k \in B : k.id = r.id)\}$, and $A \cup B$ to designate $\{r|r \in R \land ((\exists k \in A : k.id = r.id) \lor (\exists k \in B : k.id = r.id))\}$

Resource management is about managing access to scarce resources, i.e. resources for which the demand may exceed their supply. If the total requirement for a resource never exceeds its capacity, then the management is trivial: we can always provide access to the resources. For example, if a processor is used by a single task, then no scheduling is needed. We may therefore ignore resources for which the total requirement never exceeds their capacity.

1.1.2 Single-unit resources

Single-unit resources are a special case of multi-unit resources.

**Definition 1.2 (Single-unit resource).** A single-unit resource $r \in R$ is a multi-unit resource such that $r.N = 1$

A single-core processor is an example of a single-unit resource. Since only a single task can be using a processor at a time, a single-unit resource can be regarded as a multi-unit resource with capacity 1. Most literature on real-time systems focuses on single-unit resources.

In the remainder of the thesis, when we do not mention single-unit resources explicitly, we assume multi-unit resources, unless explicitly stated otherwise.

**Homogeneous vs. heterogeneous resources**

We can group resources into homogeneous resources, exhibiting the same properties, and heterogeneous resources, exhibiting different properties. Identical multiprocessors are an example of homogeneous resources, while a processor, a network interface and a bus...
found in a traditional computer are examples of heterogeneous resources. While
each resource in a heterogeneous set is modeled by an individual single-unit re-
source, we have a choice of how to model resources in a homogeneous set. We
can treat them individually or group them in a multi-unit resource, depending
on the application needs.

1.1.3 Preemptive vs. non-preemptive resources

We can classify all resources, both multi-unit and single-unit, in one of two
categories:

Definition 1.3 (Non-preemptive resource). The usage (or ownership) of a non-
preemptive resource may not be preempted without the risk of corrupting the state
of the resource.

Definition 1.4 (Preemptive resource). The usage (or ownership) of a preemp-
tive resource can be preempted without corrupting the state of the resource.

A bus is an example of a non-preemptive resource, as an ongoing message
transfer cannot be preempted without losing the message. A processor is an
example of a preemptive resource, as the state of a running task can be saved
upon a preemption and later restored.

Preemption is usually provided on the software level, by the operating sys-
tem. For example, when an interrupt arrives, the operating system first stores
the processor state, then executes the interrupt handler, and finally restores
the saved processor state. Note that actually the usage of nearly any resource
can be preempted: e.g. memory space (usually considered a non-preemptive
resource) can be “switched out” to a memory higher in the memory hierarchy
(e.g. data can be moved from the processor cache to the RAM). However, this
will come at the cost of a large performance penalty. A non-preemptive resource
is basically one for which the system designer has decided that its preemption
overhead is too large.

Notice that preemptivness is orthogonal to mutual-exclusion. Each unit of
both a preemptive and non-preemptive resource can be accessed by at most one
task at a time. For example, at most one task can be executing on each core in
a multicore processor can be accessed by at most one task at a time.

Let $N$ be the set of all non-preemptive resources, and $P$ the set of all pre-
emptive resources. As every resource is either preemptive or non-preemptive,
we have

$$(N \cup P = R) \land (N \cap P = \emptyset) \quad (1.1)$$

1.2 Application model

Let $T = \mathbb{R}_0^+$ be the time domain (of non-negative real numbers), with $t \in T$
representing a time instant.

The basic building blocks in our application model are called budgets (Sec-
tion 1.2.1), which express an execution time requirement for a particular set
of resources. These budgets can be combined sequentially to comprise periodic
components (Section 1.2.2).
1.2.1 Budgets

**Definition 1.5 (Budget).** Let \( B \subseteq T \times 2^R \) be a set of budgets. A budget \( b \in B \) describes the requirement for an amount of time on a set of resources, and is specified by a tuple \((E, R)\), which represents a budget with execution time (or time capacity) \( b.E > 0 \) on the set of budget items \( b.R \), where each budget item \( r \in b.R \) represents a requirement for (or provision of) \( r.N > 0 \) units of the resource with id \( r.id \). Moreover, all budget items in \( b.R \) must be provided simultaneously.

**Intuition behind budgets**  Budgets specify a requirement for the resource.
When a budget is scheduled on a resource during runtime, we say that the budget is *using* the resource, or that it *owns* the resource. The ownership of a resource may change, however, at any point in time each unit of a multi-unit resource may be used by at most one budget. Figure 1.1 shows an example of a budget using a single-unit resource.

![Figure 1.1: Example of a mapping of a budget on a single-core resource.](image)

**Notation**  If all budget items \( r \in b.R \) require a single unit of a resource, we may omit specifying the number of required units. For example, instead of writing \( b.R = \{(r_1, 1), (r_2, 1), (r_3, 1)\} \) we may write \( b.R = \{r_1, r_2, r_3\} \).

The mapping of budgets onto a multi-unit resource \( r \) can be visualized as a set of \( r.N \) “tracks”, with each track representing the usage of a single unit of resource \( r \), as shown in Figure 1.2. We use a dot notation to refer to the individual units in a multi-unit resource.

![Figure 1.2: Example of a mapping of budgets on resources on a platform containing a 4-core CPU and a single DSP.](image)

Figure 1.2 may represent a schedule of budgets \( b, c, \) and \( d \) on resources CPU and DSP, with \( b \) arriving at time 0, \( d \) arriving at time 1 and \( c \) arriving at time...
5 and preemption budget $d$. The arrival pattern of a budget is determined by the component it belongs to, which is discussed in the next section.

A resource manages its units internally, meaning that we cannot specify a requirement for a particular unit of a multi-unit resource. A budget item can only specify a requirement for a certain number of arbitrary units within a multi-unit resource. However, we assume that while a budget item is “executing” on a resource unit the budget item will not be migrated to another unit. For example, in Figure 1.2, once $d$’s requirement for 2 units of cpu is granted, $d$ will own the same memory blocks throughout its execution in the time interval $[1,5]$. Note that the cpu resource may decide to migrate $d$ to different units when it resumes at time 13.

Our model distinguishes between 4 single-core processors (modeled as 4 single-unit resources) and a single 4-core processor (modeled as a single multi-unit resource). In the 4 single-core processors case, we can specify a requirement for particular single-core processors, while in the 4-core case we can only specify a requirement for a particular number of cores.

During runtime, whenever a budget is using a resource its remaining execution time (or remaining time capacity) is decremented at a uniform rate.

**Definition 1.6.** Let $\beta : T \times B \to T$ be a function which keeps track of the remaining time of budgets during runtime, where $\beta(t, b)$ is the remaining time of budget $b$ at time $t$.

The remaining time should never exceed the budget’s capacity, i.e.

$$\forall t \in T, b \in B : 0 \leq \beta(t, b) \leq b.E.$$ 

(1.2)

**Definition 1.7** (Budget requirements graph). The requirements of all budgets can be represented by a budget requirements graph $G = (V,E)$ where the set of vertices $V = R \cup B$ is the union of resources and budgets, and $E$ is the set of edges, which are two-element subsets of $V$, representing the resource requirements of budgets, i.e.

$$\{a, b\} \in E \iff (b \in B \land a \in b.R) \lor (a \in B \land b \in a.R).$$

(1.3)

The graph is tripartite, as we can divide $E$ into two disjoint sets $E^P$ and $E^N$, such that

$$\forall \{a, b\} \in E^P : (a \in P \land b \in B) \lor (a \in B \land b \in P),$$

(1.4)

$$\forall \{a, b\} \in E^N : (a \in N \land b \in B) \lor (a \in B \land b \in N).$$

(1.5)

An example of a budget requirements graph is shown in Figure 1.3.

**Notation** In the budget requirements graphs we draw budget nodes as rectangles and resource nodes as circles.

**Definition 1.8.** Let $\kappa : R \to 2^B$ be a function, where $\kappa(r) = B$ means that resource $r$ is required by budgets $b \in B$, i.e.

$$\kappa(r) = \{b \in B | r \in b.R\}.$$
Figure 1.3: Example of a budget requirements graph, for a platform consisting of resources \( P = \{p_1, p_2, p_3\} \) and \( N = \{n_1, n_2\} \), and an application consisting of budgets \( B = \{a, b, c\} \), with \( a.R = \{p_1, n_1\} \), \( b.R = \{p_2, n_1, n_2\} \), \( c.R = \{p_3, n_2\} \).

1.2.2 Components

**Definition 1.9 (Component).** Let \( C \subseteq \mathbb{N} \times T \times T \times B^+ \) be the set of all components in the system. A component \( c \in C \) is specified by a tuple \((\pi, T, O, D, R)\), where \( c.\pi \) is a fixed and unique priority, \( c.T \) is its period (representing the minimum time interval between two consecutive arrivals), \( c.O \) is its initial offset (or phasing), \( c.D \) is its relative deadline, \( c.R \) is a non-empty sequence of required budgets.

Priorities are used for resolving conflicts during runtime when more than one budget tries to access a shared resource. All budgets belonging to component \( c \) share its priority \( c.\pi \). We order priorities in descending order, where \( c.\pi < d.\pi \) means that component \( c \) has a higher priority than component \( d \). We also assume that components are indexed by decreasing priority, i.e., a lower index implies higher priority. Let \( \pi_{\perp} \) be a priority lower than the lowest priority, and \( \pi_{\top} \) be a priority higher than the highest priority among all components.

**Definition 1.10.** We define \( \text{prev} : B \to B \) to be a function, where \( \text{prev}(b) \) refers to the budget preceding \( b \) in a budget sequence. More precisely, given a sequence \((b_1, b_2, \ldots, b_n)\) of \( n \) budgets, for \( i > 1 \)

\[
\text{prev}(b_i) = b_{i-1}
\]  

Notice that for \( i = 1 \), i.e. the first budget in the sequence, the function \( \text{prev} \) is not defined.

Let \( a_{c,k} \) be the \( k \)th arrival of component \( c \) (or more precisely the arrival time of its \( k \)th instance), where \( a_{c,k} \geq a_{c,k-1} + c.T \), with \( a_{c,0} \geq c.O \). At time \( a_{c,k} \) all budgets \( b \in c.R \) are replenished to their original time capacity. These budgets are dispatched sequentially, i.e., the \( k \)th instance of budget \( b \) may start executing only after the \( k \)th instance of budget \( \text{prev}(b) \) has finished. All budgets in \( c.R \) must be provided to \( c \) before time \( a_{c,k} + c.D \), i.e. the \( k \)th instance of the last budget in the required sequence \( c.R \) must finish its execution before \( a_{c,k} + c.D \). We assume \( c.D \leq c.T \).

**Definition 1.11.** We define \( \gamma : B \to C \) to be a function, where

\[
\gamma(b) = c \iff b \in c.R
\]

We say that \( c \) is the parent component of its required budget \( b \).
1.2 Application model

**Notation** For each budget we will use the shorthand notation $b.p$, $b.T$, and $b.D$ to refer to its parent component’s priority $\gamma(b).p$, period $\gamma(b).T$, and deadline $\gamma(b).D$, respectively.

**Definition 1.12.** We define $\text{index} : \mathcal{B} \rightarrow \mathbb{N}$ to be a function, where $\text{index}(b)$ refers to the index of the budget $b$ in a budget sequence. More precisely, given a sequence $(b_1, b_2, \ldots, b_n)$ of $n$ budgets,

\[
\text{index}(b_i) = i \quad (1.7)
\]

Notice that we start indexing budgets at 1.

**Definition 1.13.** We define $R[i]$ with $1 \leq i \leq n$ as the $i^{\text{th}}$ budget in the sequence of $n$ budgets $R = (b_1, b_2, \ldots, b_n)$.

**Definition 1.14.** Let $f_{b,k}$ be the finalization time of the $k^{\text{th}}$ instance of budget $b$. We define $R : \mathcal{B} \times \mathbb{N} \rightarrow \mathcal{T}$ to be a function where $R(b, k)$ is the response time of the $k^{\text{th}}$ instance of budget $b$ measured since the arrival of its parent component $\gamma(b)$, defined as $R(b,k) = f_{b,k} - a_{\gamma(b),k}$.

**Definition 1.15.** We define $WR : \mathcal{B} \rightarrow \mathcal{T}$ to be a function where $WR(b)$ it the worst-case response time of budget $b$.

Ignoring resources used within a single component In our resource and application models we consider only resources which are shared by different components. In case a resource is required only by budgets belonging to the same component, no arbitration for this resource will ever be required, since budgets belonging to the same component execute sequentially. We therefore ignore such resources in our model.

1.2.3 Comparison with the traditional task model

A task describes the work triggered as a response to an event. It may be triggered by the expiration of a timer, an external event (e.g. arrival of a network packet), or by other tasks (e.g. completion of the decoding of a video frame). Examples of tasks are decoding a video frame or transmitting a network packet.

In the real-time literature, a sporadic task $\tau_i = (\pi, T, O, D, E)$ has a unique priority $\pi$, a minimum job inter-arrival time $\tau_i.T$ (also called its period), an initial offset $\tau_i.O$ (also called its phasing), a relative deadline $\tau_i.D$, and a resource requirement $\tau_i.E$ specified by a non-empty sequence of subtasks\(^1\). In our model, a task can be expressed as a component with an empty provision interface, with the subtasks corresponding to the required budgets. More formally, task $\tau_i$ can be mapped to a component $c_j \in \mathcal{C}$, where

\[
c_j = (\pi, \tau_i, T, \tau_i.O, \tau_i.D, \tau_i.E) \quad (1.8)
\]

In the remainder of this thesis we will use the task notation when it is more convenient.

\(^1\)Most real-time literature considers tasks with a single subtask requiring a single processor resource. There are also examples of a generalization to Directed Acyclic Graphs Bril et al. (2009) of subtasks. For ease of presentation, however, we picked a sequence model.
Critical sections  In case a task requires shared (non-preemptive) resources, it can be represented as a sequence of critical and non-critical sections. A critical section requires non-preemptive resources, while a non-critical section does not. For each task Gai et al. (2001); Buttazzo (2004); Nemati et al. (2010) specify a set of critical sections guarding the access to the shared resources, where $z_{i,j}$ represents the $j$th critical section in task $\tau_i$. For each critical section their model specifies its worst-case length and the resource which is accessed.

Nested critical sections are not modeled directly. A critical section $z_{i,k}$ is nested inside $z_{i,j}$ if $z_{i,k}$ starts executing before $z_{i,j}$ has completed. Nested critical sections may lead to deadlocks, if they are not handled correctly. In multiprocessor systems, where several tasks can execute simultaneously, avoiding deadlocks due to critical sections becomes even more difficult than in uniprocessor systems. Some synchronization algorithms therefore do not allow nested critical sections. For example, they are allowed in SRP and for local resources in MSRP, and are not allowed in Priority Inheritance Protocol (PIP)\(^2\) and for global resources in MSRP.

In uniprocessor systems, one way of avoiding deadlocks due to nested critical sections is to lock them all at once. It can be visualized as stretching the inner critical sections outward, until they overlap with the outermost one. This stretching is safe, in the sense that it preserves the integrity of the resources guarded by inner critical sections. Note that, locking nested critical sections at once may result in lower schedulability of higher priority tasks, as stretching critical sections may increase the blocking time.

In our model, budget items belonging to the same budget can be regarded as a nested critical sections which have been stretched outward. In Chapter 2 we exploit this model and present a synchronization algorithm for parallel processor systems, which avoids deadlocks while allowing global nested critical sections.

1.3 Mapping

In the previous sections of this chapter we have defined the static models describing the resources and the application. Now we move on to the mapping of resources to applications. The mapping is responsible for time sharing the access to resources during runtime, commonly referred to as allocation and scheduling. An allocation assigns budgets to resources, while a schedule describes when budgets execute on their assigned resources. In this thesis we assume static allocation of budgets to resources, which is part of the budget specification.

The schedule needs to maintain budget’s timing constraints, precedence constraints (between budgets belonging to the same component), and mutually exclusive access to shared resources. All these constraints can be resolved offline, giving rise to a time-driven schedule. Alternatively, the scheduling can be done online, where scheduling decisions are event-driven and performed according to a predefined scheduling policy, with or without the cooperation from budgets (e.g. by means of calls to mutual exclusion primitives). In this thesis we consider the latter, focusing on fixed-priority preemptive scheduling and fixed-priority deferred preemption scheduling.

\(^2\)In some circumstances PIP can support nested critical sections, e.g. when they are nested in the same order in all tasks using them.
1.3 Mapping

1.3.1 Scheduling

Definition 1.16 (Schedule). A schedule $\sigma : T \times B \times R \rightarrow N$ is a function expressing resource ownership by budgets at different moments in time, where $\sigma(t, b, r) = n$ means that at time $t$ budget $b$ executes on $n$ units of resource $r$, where $r \in b.R$ and $n \leq r.N$.

Saying that “at time $t$ budget $b$ executes on $n$ units of resource $r”$ is synonymous to saying that “at time $t$ budget $b$ owns $n$ units of resource $r”$ or that “at time $t$ budget $b$ is scheduled on $n$ units of resource $r”. A component $c$ is said to be scheduled on resource $r$ when one of its budgets is scheduled on $r$.

The schedule must satisfy the following criteria:

- The scheduling function never schedules more units than a resource can provide, i.e.
  \[
  \forall t \in T, r \in R : \sum_{b \in B} \sigma(t, b, r) \leq r.N \land r \in b.R.
  \]  
  (1.9)

Note that several budgets may be scheduled on a resource at the same time, as long as together they do not require more units than the resource can provide.

- The scheduler schedules only budgets with remaining execution time, i.e.
  \[
  \forall t \in T, b \in B : (\exists r \in R : \sigma(t, b, r) > 0) \Rightarrow \beta(t, b) > 0.
  \]  
  (1.10)

- A budget is scheduled on all of its required resources, or not at all, i.e.
  \[
  \forall t \in T, b \in B : (\exists r \in R : \sigma(t, b, r) > 0) \Rightarrow (\forall r \in b.R : \sigma(t, b, r) > 0).
  \]  
  (1.11)

Definition 1.17. Let $\nu : T \times B \times R \rightarrow \{0, 1\}$ be a function, returning 1 if at time $t$ budget $b$ is scheduled on resource $r$, and 0 otherwise, i.e.

\[
\nu(t, b, r) = \begin{cases} 
1 & \text{if } \sigma(t, b, r) > 0, \\
0 & \text{otherwise.} 
\end{cases}
\]  

(1.12)

A component $c$ replenishes all its required budgets periodically, with period $c.T$. Each budget is consumed at a uniform rate whenever it is scheduled, i.e.

\[
\forall t \in T, b \in B, r \in b.R : \beta(t, b) = \left[ \frac{t}{\gamma(b).T} \right] b.E - \int_0^t \nu(x, b, r)dx.
\]  

(1.13)

The scheduling function must satisfy the realtime constraints, by making sure that every budget receives its worst-case execution time on its required resources before its parent component’s deadline, i.e.

\[
\forall k \in \mathbb{N}, b \in B, r \in b.R : (k + 1)b.E \leq \int_0^{[k+\gamma(b).T+\gamma(b).D} \nu(x, b, r)dx
\]  

(1.14)

Definition 1.18. Let $\alpha : T \times C \rightarrow B$ be a partial function, where $\alpha(t, c) = b$ means that at time $t$ budget $b$ is the current budget of component $c$, i.e.

\[
\alpha(t, c) = b \equiv b \in c.R \land \beta(t, b) > 0 \land \\
(\forall i : 1 \leq i < \text{index}(b) : \beta(t, c.R[i]) = 0) \land \\
(\forall i : \text{index}(b) < i \leq |c.R| : \beta(t, c.R[i]) = c.R[i].E)
\]  

(1.15)
The current budget $\alpha(t, c)$ of component $c$ represents the budget which $c$ is requiring at time $t$. The definition implies that at all times each component has at most one current budget. Notice that a current budget is not necessarily executing at time $t$.

**Definition 1.19.** For each resource the scheduler maintains a preemption threshold, defined by the function $\theta : T \times R \rightarrow \mathbb{N}$. A budget $b$ is allowed to start executing on resource $r$ at time $t$ only if its parent component’s priority is higher than $r$’s preemption threshold, i.e. if $\gamma(b) \pi < \theta(t, r)$.

In traditional uniprocessor scheduling with independent tasks, the preemption threshold corresponds to the priority of the running task, which can be preempted only by tasks with a higher priority. The preemption threshold can be regarded as a scheduling aid for optimizing some scheduling criterion. For example, in case of SRP on a single processor platform, the preemption threshold of the processor corresponds to the system ceiling maintained by the SRP protocol.

When a nonpreemptive resource is used by a budget, the preemption threshold of the resource is raised to the highest priority, i.e.

$$\forall t \in T, r \in \mathcal{N} : \exists b \in \mathcal{B} : \sigma(t, b, r) > 0 \Rightarrow \theta(t, r) = 1.$$  \hspace{1cm} (1.16)

The system must maintain the preemption threshold, i.e.

$$\forall t \in T, b \in \mathcal{B}, r \in \mathcal{R} : \sigma(t, b, r) > 0 \Rightarrow \theta(t, r) \leq \gamma(b) \pi$$  \hspace{1cm} (1.17)

**1.4 Platform**

The platform provides two synchronization primitives: lock($R$) and unlock($R$), for $R \subseteq \mathcal{R}$. The lock($R$) call blocks until all resource requirements in $R$ can be met. The implementation of the blocking (whether it releases the processor or spinlocks) depends on the particular scheduling algorithm. The unlock($R$) call releases all the resources in $R$. These primitives allow a budget to acquire mutually exclusive access to several resources at the same time. In particular, at the beginning and the end of a budget $b$ the corresponding resources are acquired and released using lock($b, R$) and unlock($b, R$).

From a resource management perspective (i.e. ignoring the computation details), we can write down the pseudo code for a budget $b$ as

$$lock(b, R); body(b); unlock(b, R)$$

Similarly, the program for a component requiring a sequence of budgets $\langle b_1, b_2, \ldots, b_n \rangle$ can be written down as:

$$lock(b_1, R); body(b_1); unlock(b_1, R);$$
$$lock(b_2, R); body(b_2); unlock(b_2, R);$$
$$\ldots$$
$$lock(b_n, R); body(b_n); unlock(b_n, R)$$
Chapter 2

Multi-resource management

Modern computers consist of several processing units, connected via one or more interconnects to a memory hierarchy, auxiliary processors and other devices. The traditional approach to sharing such platforms between several applications treats the whole machine as a whole: the task having access to the processor has also implicitly access to all other resources, such as a bus, memory, or network. Consequently, only a single task is allowed to execute at a time. On the one hand, this approach avoids the overheads and complexity of fine-grained scheduling of multiple resources. On the other hand, it prevents tasks with independent resource requirements to execute concurrently and thus use the available resources more efficiently. For example, a video processing task requiring the processor and operating on the processor’s local memory could in theory execute concurrently with a DMA transfer task moving data between the global memory and the network interface.

With the advent of multiprocessor platforms new scheduling algorithms have been devised aiming at exploiting some of the available concurrency, however, they usually abstract from the intricate dependencies involved in sharing common interconnects or memories. Many hardware platforms, especially in the embedded systems domain, provide mechanisms to manage the hardware resources individually. These mechanisms provide an opportunity for fine-grained scheduling and more efficient utilization of the available concurrency.

In resource constrained embedded systems a fine-grained approach is desired, which we call multi-resource scheduling.

**Definition 2.1 (Multi-resource scheduling).** A multi-resource scheduling problem deals with mapping a task set onto a set of heterogeneous resources, where each subtask requires a particular subset of resources.

Multi-resource scheduling is related to parallel-task scheduling (or simply parallel scheduling) which was originally investigated in the context of large mainframe computers (Ousterhout, 1982). When threads belonging to the same task execute on multiple processors and communicate via shared memory, then it is often desirable to schedule these threads at the same time (called gang scheduling), in order to avoid invalidating the shared memory (e.g. L2 cache) by switching in threads of other tasks. Parallel scheduling is especially desired in data intensive applications (e.g. multimedia processing), where multithreaded tasks operate on the same data (e.g. a video frame), performing different func-
Multi-resource management

tions at the same rate (Anderson and Calandrino, 2006). Parallel scheduling also shortens the time that shared data resides in caches, decreasing the chance that it will be invalidated by interfering tasks.

Multi-resource scheduling can be regarded as partitioned parallel-processor scheduling with shared resources. In partitioned parallel scheduling each task is allocated to a particular subset of processors, unlike global scheduling where a task specifies only a number of (homogenous) processors it requires, which the system allocates during runtime. While in multi-processor scheduling a task represents an execution requirement for one of several processors, in parallel-processor scheduling a task represents a requirement for simultaneous execution on several processors. Similar to multi-processors, several tasks may be executing at the same time in the system, as long as mutually exclusive access to processors or shared resources is maintained.

2.1 Problem description

Fine-grained multi-resource scheduling presents a challenge of exploiting the resources efficiently while minimizing the scheduling complexity. In this chapter we address the problem of preserving real-time constraints while exploiting the available concurrency.

Let $U(c)$ be the utilization of component $c$. For a periodic component with period $c.T$ and execution time $c.E$, utilization is given by $U(c) = \frac{c.E}{c.T}$. A simple approach to scheduling a set of components $C$ on a platform comprised of several resources is to treat the whole platform as a whole, and consequently have $\sum_{c \in C} U(c) \leq 1$. Our goal is to provide a scheduling algorithm for the multi-resource scheduling problem which meets the real-time constraints while exploiting concurrency, i.e. where $\sum_{c \in C} U(c) > 1$.

2.1.1 Contributions

In this chapter we present a novel scheduling algorithm called Parallel-SRP (PSRP) and the corresponding schedulability analysis for the multi-resource scheduling problem. We show that the algorithm does not suffer from deadlock and derive the maximum bound on blocking. We use this blocking bound as a basis for a schedulability test.

2.2 Related work

2.2.1 Multi-resource management

In this section we summarize the existing work related to multi-resource scheduling. We classify it in two dimensions: task model and scheduling problem, as shown in Table 2.1. The dependent task model describes applications composed of tasks with precedence or mutual exclusion constraints. Each task requires one processor at a time (or generally one preemptive resource). In the parallel task model application tasks require several processors concurrently (or generally several preemptive resources) and are independent, in the sense that they do not share non-preemptive resources. The multiprocessor scheduling problem deals with scheduling tasks which require an arbitrary subset of homogenous resources.
resources, while in the multi-resource scheduling problem tasks require a particular set of heterogeneous resources.

<table>
<thead>
<tr>
<th>Dependent tasks</th>
<th>Parallel tasks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dijskstra (1964, 1982), Rajkumar et al. (1988), Gai et al. (2001), Gai et al. (2003), Block et al. (2007), Brandenburg and Anderson (2008b), Lakshmanan et al. (2009)</td>
<td></td>
</tr>
<tr>
<td><strong>Multiresource</strong></td>
<td></td>
</tr>
<tr>
<td>Rajkumar et al. (1998), Saewong and Rajkumar (1999), Gopalan and Chiu (2002)</td>
<td></td>
</tr>
</tbody>
</table>

Table 2.1: Summary of the related work.

**Multiprocessor scheduling of independent tasks**

Most of the existing literature on multiprocessor scheduling can be classified in two categories: partitioned scheduling, where tasks are assigned to a particular processor, and global scheduling, where tasks or jobs can migrate between processors. Calandrino et al. (2007); Easwaran et al. (2009); Lipari and Bini (2010) present a clustering approach, which generalizes partitioned and global scheduling. Given a platform with \( m \) processors, they define a cluster as a set of \( m' \) processors, where \( 1 \leq m' \leq m \). Tasks are partitioned into clusters and globally scheduled within the clusters. A physical cluster of size \( m' \) is statically mapped to a set of \( m' \) processors, while a virtual cluster is mapped to processors dynamically such that at any time the at most \( m' \) tasks within a cluster are executing on the \( m \) processors.

**Multiprocessor scheduling of dependent tasks**

When tasks share non-preemptive resources, they may block when the resource they are trying to access is already locked by another task. In priority-based scheduling this may lead to priority inversion when a higher priority task is blocked on a resource locked by a lower priority task. In real-time systems it is important to bound the duration of priority inversion. Sha et al. (1990) investigate uniprocessor systems where tasks share several single-unit non-preemptive resource and propose the Priority Inheritance Protocol (PIP) for bounding priority inversion, and Priority Ceiling Protocol (PCP), which also avoids deadlock. The Stack Resource Policy (SRP) by Baker (1991) avoids deadlock, bounds the priority inversion to a single critical section and allows all tasks to share a single stack in systems scheduled according to fixed or dynamic priority and where tasks share multi-unit non-preemptive resources.

In multiprocessor scheduling of dependent tasks each task requires one of the available processors and may share additional logical resources with other tasks. Scheduling with such dependencies requires multiprocessor synchronization protocols.
Dijkstra (1964, 1982) presents one of the earlier synchronization protocols for multiprocessors, called the Banker's algorithm. The algorithm focuses on avoiding deadlock when several concurrently executing tasks share a common multi-unit non-preemptive resource, but it does not provide any real-time guarantees. Tasks do not have priorities nor timing constraints (besides terminating in finite amount of time), and each task is assumed to execute on its own processor. They may acquire and release the units of the shared resource in any order, as long as the total number of claimed units does not exceed a specified maximum, and as long as they release all claimed units upon completion. Habermann (1969) presents a generalization of the Banker's algorithm to several non-preemptive multi-unit resources.

More recently, existing real-time synchronization protocols for uniprocessors have been extended to multiprocessors. Rajkumar et al. (1988) present the Multiprocessor Priority Ceiling Protocol (MPCP) and Distributed Priority Ceiling Protocol (DPCP) (for distributed memory systems). Gai et al. (2001) present the Multiprocessor Stack Resource Policy (MSRP) and show that it outperforms MPCP. All three protocols assume partitioned EDF scheduling. The Flexible Multiprocessor Locking Protocol (FMLP) by Block et al. (2007) is an example of a resource sharing protocol for both partitioned and global EDF multiprocessor scheduling.

In MPCP, DPCP, MSRP and FMLP, depending on the task allocation (i.e. task partitioning), a local resource is one which is used only by tasks assigned to the same processor, while a global resource is used by tasks belonging to different processors. The MPCP, DPCP and MSRP protocols focus on sharing global resources, assuming access to local resources is synchronized using existing uniprocessor protocols (the PCP and SRP protocols).

MPCP, DPCP, MSRP and FMLP assign ceilings to locks in order to defer requests which otherwise could be granted. Deferring the requests allows to reduce and bound the blocking due to priority inversion. There are two approaches for dealing with blocking on multiprocessors: spin-lock based and suspension based. When a spin-lock is used, a task blocking on a locked global resource continues to spin on the resource, preventing lower priority tasks from executing. In the suspension approach, when a task blocks on a global resource it is suspended, enabling lower priority tasks to execute. The key benefit of the suspension approach is that the idle time during suspension is available for execution by other tasks. The disadvantages, however, are the penalties of back-to-back executions and multiple priority inversions per task.

The original description of MPCP is suspension based, while the MSRP is spin-lock based. Gai et al. (2003) compare MPCP and MSRP and conclude that neither algorithm is better than the other across the whole spectrum of possible task sets. However, when global critical sections are short and access to local resources dominates access to global resource, MSRP outperforms MPCP.

Block et al. (2007) claim that FMLP outperforms MSRP, by showing that FMLP can schedule more task sets than MSRP. They simulate the schedulability of many tasks sets, where each task set specifies the task’s period, execution time, and critical sections for shared resources (including their nesting). They assume freedom in partitioning the task set, i.e. that tasks may be assigned to arbitrary processors. MRSP requires that all tasks accessing the same resources in a nested fashion be allocated to the same processor, which may lead to over-utilization of certain processors. FMLP, on the other hand, does not require

Version 20110511152800
such tasks to be allocated to the same processor, allowing in some cases to avoid
over-utilization and schedule task sets which are not schedulable under MSRP.
Arbitrary partitioning, however, may not necessarily hold for heterogeneous
systems, where different processors may provide different functionality.

Brandenburg et al. (2008); Brandenburg and Anderson (2008a) investigate
the performance penalties between various spin-lock and suspension based pro-
tocols (MPCP, DPCP and FMLP) and conclude that spin-lock based approaches
incur smaller scheduling penalty than suspension based. Lakshmanan et al.
(2009) extend the original suspension-based MPCP description with spin lock-
ing, compare the two implementations and show the opposite, i.e. that for low
preemption overheads and long critical sections the suspension-based approaches
perform better, while in other settings they perform similar.

Neither MPCP nor MSRP allow nested global critical sections. While MSRP
preserves SRP’s single-stack property for all tasks executing on the same pro-
cessor, it does not address multi-unit resources, which were supported by the

FMLP supports nested critical sections by means of resource groups. Two
resources belong to the same resource group iff there exists a task which requests
both resources at the same time. Before a task can lock a resource it must
first acquire the lock to the corresponding resource group $G(r)$. This ensures
that only a single job can access the resources in a group at any given time.

**Multiprocessor scheduling of parallel tasks**

The problem of multiprocessor scheduling of parallel tasks is also referred to
in the literature as gang scheduling. Gang scheduling was first introduced in
Ousterhout (1982), and later discussed among others in Feitelson (1990); Feitel-
son and Rudolph (1992). In its original formulation it was intended for schedul-
ing concurrent jobs on large multiprocessor systems. However, gang scheduling
has the potential to provide fine grained scheduling within an embedded plat-
form, using the available resources more efficiently than the standard approach
of treating the whole platform as a single resource.

The work on parallel tasks scheduling with real-time constraints is more
recent. Anderson and Calandrino (2006) present a homogenous multi-processor
scheduling algorithm for independent tasks which encourages individual threads
of a multi-threaded task to be scheduled together. They observe that when such
threads are cooperative and share a common working set, this method enables
more effective use of on-chip shared caches resulting from fewer cache misses.
They consider a multi-core architecture with symmetric single-threaded cores
and a shared L2 cache. They employ the global PD$^2$ and EDF schedulers. Notice
that their algorithm only “encourages” individual threads of a multithreaded
task to be scheduled together, unlike gang scheduling, which guarantees that
these threads will be scheduled together. Also, the threads belonging to the
same task may have different execution times, but a common period.

Lakshmanan et al. (2010) adopt the basic fork-join model. A task starts
executing in a single master thread until it encounters a fork construct. At that
moment it spawns multiple threads which execute in parallel. A join construct
synchronizes the parallel threads. Only the master thread can fork and join. A
task can therefore be modeled as an alternating sequence of single- and multi-
threaded subtasks. Unlike gang scheduling, each thread (including the parallel

Version 20110511152800
executing threads) can be preempted. Kato and Ishikawa (2009) present a schedulability analysis for preemptive EDF gang scheduling on multiprocessors. They assume that a task $\tau_i$ requires a subset of $m_i$ homogenous processors. Their schedulability analysis is overly pessimistic.

At a first glance the gang scheduling problem looks similar to synchronization on multiprocessors: tasks can execute on one of several processors and can share global logical resources. There are variants where tasks are assigned to a particular processor (partitioned scheduling) and where tasks or jobs can migrate between processors (global scheduling). While global scheduling deals with both allocating tasks to processors and scheduling them during runtime, partitioned scheduling assumes task allocation is done offline limiting the runtime effort to local scheduling on each processor. In both cases, however, a task executes on exactly one processor at a time. The key part of gang scheduling is that a task needs to execute on several preemptive resources (e.g. processors) or non-preemptive resources (e.g. graphical processing units) concurrently. A gang scheduler resembles a global multiprocessor scheduler in the sense that it maintains a global ready queue, while in partitioned multiprocessor scheduling each processor maintains its local ready queue and schedules tasks locally.

2.2.2 Synchronization protocols

In this section we recapitulate the Banker’s algorithm, the Stack Resource Policy (SRP), and the MSRP protocol, which inspired our PSRP algorithm described later in this chapter.

Banker’s algorithm

Dijkstra (1964, 1982) presents the Banker’s algorithm, which avoids deadlock in a system where several (concurrent) tasks share a common multi-unit resource. In its original formulation the shared resource is money which is lent out by a banker to several clients, who need the money to complete their project. Since all clients can operate in parallel, we can consider them as tasks executing on a multiprocessor platform, consisting of as many processors as there are tasks. Each client can borrow one or more florins\(^1\), which are then added to his loan, and may return one or more florins, which are then subtracted from his loan. Each client specifies the maximum loan he will ever require at a time and is obliged to return any outstanding loan upon completion of his project, which he is assumed to complete in finite amount of time. The money provided by the banker can be regarded as a non-preemptive multi-unit resource, since the money lent out to a client cannot be claimed back by the banker until the client returns it, and each florin can be regarded as a unit of the multi-unit resource. The algorithm is invoked whenever a client requests a loan. It checks if granting the client the loan will never lead to a deadlock, given the outstanding claims and loans of other clients, i.e. if it can be guaranteed that all (present and future) requests can be granted within a finite amount of time if clients current request is granted. In case a potential deadlock is detected, the request is denied, otherwise it is granted.

\(^1\)Old Dutch currency.
2.2 Related work

The algorithm maintains two variables for each client $i$: the current $loan(i)$ and the outstanding $claim(i)$. Let $need(i)$ be the maximum loan which client $i$ will ever require. At all times for each client it holds:

$$loan(i) + claim(i) = need(i).$$

Let $capital$ be the banker’s initial capital and $cash$ be the remaining money, given the outstanding loans. At any time it holds:

$$cash = capital - \sum_i loan(i).$$

The banker may not lend out more money that the initial capital, i.e.

$$cash \geq 0.$$

The banker maintains a list of clients in ascending order according to their outstanding claims, hence

$$claim(i) \leq claim(i + 1).$$

To guarantee absence of deadlock, the banker simulates actually granting the request and checks if the resulting state is safe. The algorithm for determining if the resulting state is safe is outlined in Algorithm 1. It returns $true$ if the state is safe and $false$ otherwise.

Algorithm 1 The Banker’s Algorithm

```java
available := cash;
i := 1;
while claim(i) \leq available do
    available := available + loan(i);
    if available < capital then
        i := i + 1;
    else
        return true;
end if
end while
return false;
```

While the Banker’s algorithm prevents deadlock, it does not bound the response time of tasks. It only assumes that each task has a finite computation time, and consequently it can only guarantee that each task will complete within a finite amount of time.

SRP

The Stack Resource Policy (SRP) was introduced by Baker (1991). It was intended as a priority inversion protocol for accessing shared logical multi-unit resources on a uniprocessor system. The protocol prevents deadlock, chained blocking, allows to share a single stack and has a very straightforward implementation.
The seminal paper on SRP considered a uniprocessor system. In order to support fixed-priority scheduling as well as Earliest Deadline First (EDF), Baker (1991) equipped each task \( \tau_i \) with a preemption level \( \lambda_i \). Unlike the priority of a task in EDF, the preemption level is fixed during runtime. SRP also assigns each resource a resource ceiling which is equal to the highest preemption level among all tasks accessing the resource. During runtime, the system ceiling is equal to the highest resource ceiling among the currently accessed resources.

The original SRP scheduling rule says:

**Definition 2.2** (SRP scheduling rule). A job is not allowed to start executing until its priority is the highest among the ready jobs and its preemption level is higher than the system ceiling.

Figure 2 outlines a uniprocessor fixed-priority scheduler supporting SRP based synchronization.

**Algorithm 2** Single processor FPPS scheduler based on SRP

\[
\tau_i := \text{highest priority task in the ready queue}
\]

if \( \pi_i < \text{running priority} \land \lambda_i < \text{system ceiling} \) then

switch in(\( \tau_i \))

end if

In the remainder of this thesis we will focus on fixed-priority scheduling and will use tasks priority instead of its preemption level.

**MSRP**

Gai et al. (2001) extended the SRP protocol to multiprocessors, which they call MSRP. They assume partitioned multiprocessor scheduling, meaning that each task is statically assigned to a processor\(^2\). Depending on this allocation, they distinguish between local and global resources: *local resources* are accessed by tasks assigned to the same processor, and *global resources* are used by tasks assigned to different processors. The MSRP scheduling algorithm is defined by the following five rules\(^3\):

**Rule 1**: For local resources, the algorithm is the same as the SRP algorithm. In particular, we define a preemption level for every task, a ceiling for every local resource, and a system ceiling \( \Pi_k \) for every processor \( P_k \).

**Rule 2**: Tasks are allowed to access local resources through nested critical sections. It is possible to nest local and global resources. However, it is not possible to nest global critical sections; otherwise a deadlock can occur.

**Rule 3**: For each global resource, every processor \( P_k \) defines a ceiling greater than or equal to the maximum preemption level of the tasks on \( P_k \).

**Rule 4**: When a task \( \tau_i \), allocated to processor \( P_k \) accesses a global resource \( \rho' \), the system ceiling \( \Pi_k \) is raised to \( \text{ceil}(\rho') \) making the task nonpreemptable.

Then, the task checks if the resource is free: in this case, it locks the

---

\(^2\)They present a utilization based analysis for partitioned EDF, however, the MSRP algorithm can be applied to fixed priority scheduling.

\(^3\)These rules are taken verbatim from Gai et al. (2001).
resource and executes the critical section. Otherwise, the task is inserted in a FCFS queue in the global resource, and then performs a busy wait.

**Rule 5:** When a task $\tau_i$, allocated to processor $P_k$, releases a global resource $\rho^r$, the algorithm checks the corresponding FCFS queue, and, in case some other task $\tau_j$ is waiting, it grants access to the resource, otherwise the resource is unlocked. Then, the system ceiling $\Pi_k$ is restored to the previous value.

### 2.2.3 Related work in terms of our system model

Figure 2.2 summarizes the related work presented in Section ?? using the terminology of our system model. It considers only scheduling algorithms for real-time systems.

### 2.3 Towards multi-resource sharing

Nested critical sections in multiprocessor systems are problematic, as they may lead to deadlock. There are several approaches to deal with nested *global* critical sections:

- Disallow nested global critical sections (Rajkumar et al., 1988; Gai et al., 2003)
- Allow locking of resources only in a particular order
- Claim nested resources simultaneously Block et al. (2007); Brandenburg and Anderson (2008b)
- Allow arbitrary nesting Dijkstra (1964); Habermann (1969)

Similar to the Banker’s algorithm, the original SRP protocol handles multi-unit resources, however, SRP applies only to uniprocessor systems. It follows a similar idea to the one behind the Banker’s algorithm: if a request for a resource could ever lead to deadlock, the request is denied. While the Banker’s algorithm checks for deadlock at the time the task tries to access a non-preemptive resource, SRP checks for deadlock at the time the task is activated. The resource ceilings in SRP, which are computed offline, are basically a very efficient encoding of the deadlock conditions. During runtime, whenever a task requests a resource, the scheduler avoids deadlock by simply checking if the task’s priority is higher than the system ceiling. Priority based scheduling together with a known priority assignment policy (e.g Rate Monotonic or Deadline Monotonic) and timing properties of tasks, allow to also provide real-time guarantees. Therefore, SRP can be regarded as an efficient implementation of the Banker’s algorithm in the domain of priority-based uniprocessor scheduling with shared resources and real-time constraints.

Following a similar reasoning one may consider MSRP as an efficient implementation of the Banker’s algorithm in the domain of priority-based multiprocessor scheduling with shared resources and real-time constraints. In the remainder of this chapter we present a scheduling algorithm for fixed-priority parallel-processor scheduling with shared resources and real-time constraints, which was inspired by the Banker’s algorithm, SRP and MSRP.
<table>
<thead>
<tr>
<th>Description</th>
<th>Multi-resource</th>
<th>Priorities</th>
<th>( P )</th>
<th>( N )</th>
<th>( b.R )</th>
<th>( c.R )</th>
<th>Nested global critical sections</th>
</tr>
</thead>
<tbody>
<tr>
<td>Banker’s algorithm</td>
<td>P, G</td>
<td>F, D</td>
<td>( {p, k</td>
<td>k = 1}^3 )</td>
<td>( {n, k}k = N )</td>
<td>( {p, 1}) or ( {p, 1, (n, k)}) ( k \leq N )</td>
<td>( N^3 )</td>
</tr>
<tr>
<td>Habermann (1969)</td>
<td>P, G</td>
<td>F, D</td>
<td>( {p, k</td>
<td>k = 1}^3 )</td>
<td>( {n, k}k = N )</td>
<td>( {p, 1}) or ( {p, 1, (n, k)}) ( k \leq N )</td>
<td>( N^5 )</td>
</tr>
<tr>
<td>SRP</td>
<td>-1</td>
<td>F, D</td>
<td>( {p, k}) ( k = 1 )</td>
<td>( {n, k}k = N )</td>
<td>( {p, 1}) or ( {p, 1, (n, k)}) ( k \leq N )</td>
<td>( N )</td>
<td>-</td>
</tr>
<tr>
<td>MSRP</td>
<td>P</td>
<td>D</td>
<td>( {p, k</td>
<td>k = 1} )</td>
<td>( {n, k}k = N )</td>
<td>( {p, 1}) or ( {p, 1, (n, 1)})</td>
<td>( N )</td>
</tr>
<tr>
<td>FMLP</td>
<td>P, G</td>
<td>D</td>
<td>( {p, k</td>
<td>k = 1} )</td>
<td>( {n, k}k = N )</td>
<td>( {p, 1}) or ( {p, 1, (n, 1)})</td>
<td>( N )</td>
</tr>
<tr>
<td>Brandenburg and Anderson (2008b)</td>
<td>P, G</td>
<td>F, D</td>
<td>( {p, k</td>
<td>k = 1} )</td>
<td>( {n, k}k = N )</td>
<td>( {p, 1}) or ( {p, 1, (n, 1)})</td>
<td>( N )</td>
</tr>
<tr>
<td>Anderson and Calandrino (2006)</td>
<td>P</td>
<td>G</td>
<td>( {p, N})</td>
<td>( \emptyset )</td>
<td>( {p, N^2})</td>
<td>( N^6 )</td>
<td>-</td>
</tr>
<tr>
<td>Kato and Ishikawa (2009)</td>
<td>-1</td>
<td>D</td>
<td>( {p, N})</td>
<td>( \emptyset )</td>
<td>( {p, k}) ( k \leq N )</td>
<td>( 1 )</td>
<td>-</td>
</tr>
<tr>
<td>Goossens and Berten (2010)</td>
<td>-1</td>
<td>F</td>
<td>( {p, N})</td>
<td>( \emptyset )</td>
<td>( {p, k}) ( k \leq N )</td>
<td>( 1 )</td>
<td>-</td>
</tr>
<tr>
<td>PSRP</td>
<td>P</td>
<td>F</td>
<td>( {p, k</td>
<td>k = 1} )</td>
<td>( {n, k}k = N )</td>
<td>( {p, k</td>
<td>k = 1}) ( \cup {n, k}k \leq N )</td>
</tr>
</tbody>
</table>

P Partitioned  
G Global  
F Fixed priority  
D Dynamic priority (EDF)

1. It does not make sense to talk about the global or partitioned scheduling, since there is a single resource.  
2. Budget items do not have to be provided simultaneously (they are “encouraged” to be scheduled together).  
3. Each component is executing on its own processor.  
4. Group lock for nested global critical sections.  
5. Budgets may be nested (rather than only sequential).  
6. All budgets have the same length (the scheduling quantum), and all budgets belonging to the same components have a requirement for the same number of units of the multi-unit processor.

Table 2.2: Summary of the related work
2.4 Parallel-SRP (PSRP)

In this section we restrict our model and allow multi-unit resources to be only non-preemptive. We require that all preemptive resources are single-unit, i.e.

\[ \forall r \in \mathcal{P} : r.N = 1 \]  \hspace{1cm} (2.1)

We present the Parallel-SRP (PSRP) scheduler, which is inspired by the MSRP scheduler for multi processors and can be regarded as its generalization to the parallel-task model.

2.4.1 PSRP vs. MSRP

PSRP differs from MSRP in the following ways:

- PSRP supports nested global critical sections, while MSRP disallows nested global critical sections.
- PSRP supports multi-unit non-preemptive resources, while MSRP supports only single unit non-preemptive resources.
- PSRP allows a budget to require several preemptive resources (e.g. several processors in parallel), while under MSRP each task requires exactly one preemptive resource.

In the remainder of this section we discuss the differences in more detail.

Local vs. global resources

MSRP distinguishes between local and global resources. A resource is called local if it is accessed only by tasks assigned to the same processor, otherwise it is called global. Let us first explain the rationale behind local and global resources before adapting their definition to parallel processor systems.

In uniprocessor systems, when a task blocks on a shared logical resource, then the resource can only be released by another task running on the same processor. Therefore, the only option is to suspend the blocked task and allow the other task to continue, so that eventually the resource will be released.

In multiprocessor systems, a task can access local and global resources. Similar to uniprocessors, a local resource can only be acquired by another task running on the same processor. A global resource, however, can be acquired by a task running on a different processor. We therefore have two options for a task blocked on a global resource: we can either suspend it and allow another task assigned to the same processor to do useful work while the blocked task is waiting, or we can have the blocked task perform a spin lock (also called a “busy wait”).

In either case, the blocking time has to be taken into account in the schedulability analysis. When a task suspends on a global resource, it allows lower priority tasks to execute and lock other resources, potentially leading to priority inversion. When a task spins on a global resource, it wastes the processor time which could have been used by other tasks. Hence, for global resources it is very important to keep the resource holding times short, more important.
than for local resources. In the multiprocessor case it therefore makes sense to distinguish between local and global resources.

In our investigation of parallel processor systems we propose a generalization of the notions of local and global resources as defined by Gai et al. (2001). The essential property of a global resource is that it is required by budgets which can attempt to access their resources independently of each other (i.e. budgets which are not synchronized on a shared preemptive resources).

**Definition 2.3 (Local and global resources).** We define a resource \(r\) as local if (i) it is preemptive and accessed only by budgets which require only non-preemptive resources besides \(r\), or (ii) it is non-preemptive and accessed by budgets which also share one preemptive resource \(p\) and this \(p\) is the same for all budgets in \(\kappa(r)\). Otherwise the resource is global. More formally, we use \(R^{local}\) and \(R^{global}\) to denote the sets of local and global resources, respectively, such that

\[
R^{local} = \{r \in P|\forall b \in \kappa(r) : (\forall s \in b.R \setminus \{r\} : s \in N) \cup (r \in N|\forall b \in \kappa(r) : b.R \cap P \neq \emptyset) \land (|P \cap \bigcup_{b \in \kappa(r)} b.R| = 1)\}
\]

\[
R^{global} = R \setminus R^{local}
\]

(2.2)

Figure 2.1 illustrates the difference between local and global resources.

Notice that the local/global classification in MSRP is limited only to non-preemptive resources, while in our definition also includes preemptive resources.

**Local vs. global budgets**

Similarly to MSRP distinguishing between local and global critical sections (guarding access to local and global resources, respectively), in PSRP we distinguish between local and global budgets.

**Definition 2.4 (Local and global budgets).** We define a local budget as one requiring at least one local resource. More formally, we use \(B^{local}\) and \(B^{global}\) to denote the sets of local and global budgets, respectively, such that

\[
B^{local} = \{b \in B|\exists r \in b.R : r \in R^{local}\}
\]

\[
B^{global} = B \setminus B^{local}
\]

(2.3)

Notice that a local budget can require both local and global resources, while a global budget requires only global resources.

When identifying global and local resources and budgets, we consider only the identity of the required resources and ignore the number of required units. This means that a budget requiring any number of units of a multi-unit resource is treated the same way as a budget requiring one unit of a single-unit resource.

**Nested critical sections**

While MSRP supports only local nested critical sections. A task may acquire local non-preemptive resources individually in a nested fashion (as long as the critical sections are properly nested).

PSRP supports both local and global nested critical section by shifting the inner ones outward resulting in a similar approach to the resource groups of
2.4 Parallel-SRP (PSRP)

Figure 2.1: Example illustrating local and global resources and budgets in a budget requirements graph, for a system comprised of $P = \{p_1, p_2, p_3\}$, $N = \{n_1, n_2, n_3, n_4\}$, $B = \{a_1, b_1, c_1, c_2, d_1, d_2, e_1, f_1\}$, $C = \{a, b, c, d, e, f\}$ with $a.R = \{a_1\}$, $b.R = \{b_1\}$, $c.R = \{c_1, c_2\}$, $d.R = \{d_1, d_2\}$, $e.R = \{e_1\}$, $f.R = \{f_1\}$, and $a_1.R = \{p_1\}$, $b_1.R = \{p_1, n_1\}$, $c_1.R = \{p_1, n_1, n_2\}$, $c_2.R = \{p_2, p_3, n_2\}$, $d_1.R = \{p_3, n_2, n_3\}$, $d_2.R = \{n_3\}$, $e_1.R = f_1.R = \{p_3, n_4\}$.

FMLP (see Section 1.2.3). It relies on the lock() and unlock() operations described in Section ??, which allow budgets to acquire several resources at once. This holds for both local and global resources, i.e. under PSRP a task must acquire all nested local resources at once.

Resource requirements

MSRP is limited to tasks which require one preemptive (single-unit) resource, zero or more local non-preemptive (single-unit) resources, and at most one global non-preemptive (single-unit) resource at a time (to prevent problems with nesting). Under PSRP, each budget can require zero or more local and global preemptive (single-unit) resources and zero or more local and global (multi-unit) non-preemptive resources.

MSRP assumes that each task is assigned to exactly one processor, i.e. all of its critical and non-critical sections execute on the same processor. In our model we treat each budget (corresponding to a critical or non-critical section in MSRP) independently. Each budget can start executing on a new set of processors. In multi-processor terminology we can say that our tasks can migrate between processors at budget boundaries.

Execution model

Under MSRP, a task may lock resources only when it is already executing. This means that when a task $\tau_i$ allocated to processor $P_k$ enters a global critical section on resource $\rho^*$, raising the system ceiling of the processor $P_k$ to $\text{ceil}(\rho^*)$, task $\tau_i$ is already running, i.e. its priority is higher than the system ceiling before entering the critical section. Consequently, the system ceiling of processor $P_k$ will only be raised by $\tau_i$ when it is low enough to allow $\tau_i$ to execute in the first place.

PSRP is based on the model described in Chapter ??, where budgets explicitly require the processor (if any). The first thing a budget $b$ does after arrival is execute the lock($b.R$) operation, which locks all the resources required by $b$. 
One question is, where this lock operation is executed. Later in this chapter we will see that some budgets will require a “leading” preemptive resource, which controls the budget execution. Such budgets will execute the lock operation on that leading resource. Other budgets can execute the lock operation on any resource they require (i.e. it is up to the application developer to designate the “leading” resource for those budgets). The lock \((b,R)\) operation is executed at priority \(b.\pi\). This means that the lock \((b,R)\) operation may need to wait until a higher priority budget finishes executing.

**Fixed priority scheduling**

The MSRP protocol described by Gai et al. (2001) is intended for the Earliest Deadline First (EDF) scheduler. Similar to SRP, it relies on preemption levels, which are statically assigned to each task. For local resources, the EDF based MSRP scheduler uses a combination of tasks’ dynamic priorities (i.e. absolute deadlines) and fixed preemption levels to determine if a ready task should preempt the currently running task.

The PSRP protocol assumes a fixed priority scheduler, where priorities are static. Component priorities and preemption levels can therefore be unified. In PSRP component priorities are used instead of preemption levels.

### 2.4.2 The PSRP algorithm

The PSRP algorithm follows the following set of rules:

**Rule 1:** For local resources \(r \in R_{local}\), the algorithm is the same as the SRP algorithm. In particular, we define resource ceiling \(\varphi : R \rightarrow \mathbb{N}\), where \(\varphi(r)\) is equal to the highest priority of any component requiring \(r\). We also equip every local preemptive resource with a ready queue, which stores components waiting for or executing on the resource based on their priority, and define a ceiling for every local non-preemptive resource.

**Rule 2:** Each global resource \(r \in R_{global}\) is equipped with a First Come First Serve (FCFS) resource queue \(r.queue\). The resource queue stores components which are waiting for or executing on the resource.

**Rule 3:** When a component \(c\) attempts to lock a set of resources \(R\) using \(lock(R)\), it is inserted into the resource queues of all global resources in \(R\). Moreover, this insertion is atomic, meaning that no other component can be inserted into any of the resource queues in \(R\) before \(c\) has been inserted into all queues in \(R\).

When a component \(c\) releases a set of resources \(R\) using \(unlock(R)\), it is removed from the resource queues of all global resources in \(R\). Each \(unlock(R)\) must be preceded by a \(lock(R)\) call, with the same \(R\).

**Rule 4:** A component \(c\) is said to be ready at time \(t\) if for all resources \(r\) required by its currently active budget \(b\): the component \(c\) is at the head of \(r.queue\), its priority is higher than the preemption threshold of resource \(r\), and \(r\) has enough units available, i.e.

\[
\text{ready}(t,c) \equiv (\forall r \in \alpha(t,c).R \cap R_{global} : \text{head}(t,r.queue) = c) \wedge \\
(\forall r \in \alpha(t,c).R \cap R_{local} : c.\pi < \theta(t,r)) \wedge \\
(\forall r \in \alpha(t,c).R : r.N \leq \eta(t,r))
\]  (2.4)
where $\text{head}(t, q)$ is the head element of queue $q$ at time $t$.

**Rule 5:** If after adding a component $c$ to the queue of resource $r$ the head of the queue has changed (i.e. if $c$ ends up at the head), or if after removing a component from a resource queue the queue is not empty, then the scheduler checks if the head component is ready, according to (2.4). If so, then the component is scheduled. Otherwise, $c$ performs a spin lock (at the highest priority) on each resource containing $c$ at the head of its queue.

Notice that if a component $c$ requires several units of a resource $r$, then it will spin lock until enough of the budgets currently using $r$ have completed and released a sufficient number of units of $r$ for $c$ to continue.

**Rule 6:** The following invariant is maintained: the preemption threshold of each preemptive resource is equal to the top priority whenever a budget requiring a global resource is spinning or executing on it, i.e.

\[
\forall t \in T, p \in \mathcal{P} : (\exists b \in B^{\text{global}} : \sigma(t, b, p) > 0) \Rightarrow \theta(t, p) = \pi_+ \tag{2.5}
\]

On some resources “spinning” may not make sense (e.g. spinning on a bus). The spinning operation essentially “reserves” a resource, preventing other tasks to execute on it. It can therefore be implemented differently on different resources. Figure 2.2 shows depicts the state diagram for any resource (preemptive or non-preemptive). It distinguishes between actively “using” a resource and “reserving” it without doing actual work (e.g. spinning).

![State diagram for a resource.](image)

**Blocking vs. interference vs. waiting** A budget is *blocked* by lower priority budgets and experiences *interference* from higher priority budgets (on a local resource). The term *waiting* is used to describe competition for a global resource. A local budget will be waiting for global budgets using the same global resource only. A global budget will be waiting for both global and local budgets sharing global resources.
2.4.3 Schedulability analysis for PSRP

We first show that PSRP does not suffer from deadlock nor from live lock, and then we proceed to bound the worst-case response time of components.

Lemma 2.1. PSRP does not suffer from deadlock.

Proof. Since a lock(R) is accompanied by a corresponding unlock(R), with the same R, each locked resource will eventually be unlocked, provided no budget is blocked indefinitely. We need to show the absence of dependency cycles when budgets are waiting for resources and after they have started executing.

Let us assume a budget b is executing lock(b.R) and it needs to wait for some of the resources in b.R. Since the resource queues handle the components in FCFS order and since the addition to all queues in b.R is atomic, a dependency cycle when budget b is waiting for resources is not possible.

Since we have assumed no nested critical sections and all resources required by a budget are provided simultaneously (i.e. either all or none), once a budget starts executing it will be able to complete and release the acquired resources. Hence a deadlock cannot occur.

Lemma 2.2. PSRP does not suffer from live lock.

Proof. We need to show that every budget b will eventually start executing. Since the resource queues handle the components in FCFS order and since the addition to all queues in b.R is atomic, as long as each budget completes in finite amount of time, every budget inserted into a resource queue will have to wait for at most finite amount of time. Hence a live lock cannot occur.

To show that components will meet their real-time constraints, we derive the bound on their worst-case response time. To see if a component is schedulable we check if the worst-case response time of its last budget, measured from the arrival time of the component, is within the component’s deadline.

Lemma 2.3. A local budget requires exactly one preemptive local resource.

Proof. According to (2.2), budgets which share a local resource share exactly one preemptive resource. According to (2.3), every local budget requires at least one local resource. Lemma 2.3 follows.

According to Lemma 2.3, each local budget requires exactly one preemptive resource. This preemptive resource will dictate the behavior of the local budgets sharing it. PSRP will use the priority-ordered ready queue to schedule local budgets based on their priority. Each global budget, according to (2.3), requires only global resources. PSRP will use the resource queues attached to the global resources to schedule the global budgets non-preemptively in FCFS order. In the remainder of this section we derive an equation for the worst-case response time for local budgets and for global budgets, followed by an example illustrating the derived schedulability analysis.
2.4 Parallel-SRP (PSRP)

A global budget spin locks and executes on all its required resources at the highest priority. Consequently, as a global budget cannot be preempted, its response time is comprised of three time intervals, as illustrated in Figure 2.3.

The delay due to budgets preceding \( b \) in the \( \gamma(b).R \) sequence is equal to the response time of the previous budget, which can be computed by iterating through the sequence starting with the first budget. The execution time of budget \( b \) is simply \( b.E \). The interesting part is the time that budget \( b \) spends waiting for resources in \( b.R \).

The MSRP algorithm assumes that at any time each global budget \( b \) requires one preemptive and at most one nonpreemptive resource. Also, access to a global resource is granted to budgets in FCFS order. Consequently, they observe that the worst-case spinning time of budget \( b \) on a preemptive resource is equal to the sum of the budget execution times of all budgets sharing the nonpreemptive resource with \( b \). In our model, a budget can require an arbitrary number of preemptive and nonpreemptive resources, which may result in longer spinning time.

**Example 2.1** Consider a platform comprised of four processors \( P = \{p_1, p_2, p_3, p_4\} \), executing an application consisting of four components, each containing one budget. We name these budgets \( B = \{a, b, c, d\} \), and define their resource requirements as follows: \( a.R = \{p_1\} \), \( b.R = \{p_1, p_2\} \), \( c.R = \{p_2, p_3\} \), and \( d.R = \{p_3, p_4\} \), as shown in Figure 2.4. Notice that all resources and budgets are global.

**Legend:** \( \downarrow \) component arrival, \( \quad \) previous budgets, \( \quad \) current budget

Figure 2.3: Response time of a global budget \( b \).

**Response time of global budgets**

Figure 2.4: A budget requirements graph for a system comprised of \( P = \{p_1, p_2, p_3, p_4\}, N = \emptyset, B = \{a, b, c, d\} \) with \( a.R = \{p_1\}, b.R = \{p_1, p_2\}, c.R = \{p_2, p_3\}, \) and \( d.R = \{p_3, p_4\} \).

Let us assume a scenario, where the processors are idle and budgets \( a, b, c, d \) arrive soon after each other, as shown in Figure 2.5.
Figure 2.5: Example of parallel-chained blocking of global budgets. The figure shows the arrival and execution of budgets $B = \{a, b, c, d\}$ on preemptive resources $P = \{p_1, p_2, p_3, p_4\}$ and the contents of their resource queues. Since we assumed that each component contains only one budget, for ease of presentation we refer to the components inside the resource queues by the corresponding budget names.

When budget $a$ arrives and processor $p_1$ is idling, it is immediately scheduled and starts executing. When budget $b$ arrives, requiring processors $p_1$ and $p_2$, and encounters a busy processor $p_1$, it is added to the resource queues of $p_1.queue$ and $p_2.queue$. Since it is at the head of $p_2.queue$ it starts spinning on $p_2$ (at the highest priority). Soon after budget $c$ arrives and similarly is inserted into the resource queues of $p_2.queue$ and $p_3.queue$ and starts spinning on $p_3$. When budget $d$ arrives soon after budget $c$, it is inserted into $p_3.queue$ and $p_4.queue$ and starts spinning on $p_4$. When budget $a$ completes and releases $p_1$, it is removed from $p_1.queue$, enabling budget $b$, which starts executing. This process continues, subsequently releasing budgets $c$ and $d$. Notice that budget $d$ cannot start executing before $c$ has completed, which cannot start before $b$ has completed.

A budget may be required to wait inside of a resource queue, either passively waiting in the queue’s tail or actively spinning at its head. Example 2.1 suggests that, under PSRP, a budget may need to wait on its required resources until all budgets which it “depends on” in the budget requirements graph have
completed. We can observe, however, that budgets belonging to the same component are executed sequentially (by definition), and therefore cannot interfere with each other. The time that budget \( b \) may need to wait is therefore limited to those budgets, which \( b \) “depends on” if we ignore budgets belonging to the same component. Moreover, as we will show later, a budget depends only on other budgets which share global resources. Let us formally define the dependency relation.

Definition 2.5. A partial budget requirements graph \( G' = (V', E') \) derived from budget requirements graph \( G = (V, E) \) is a subgraph of \( G \), with \( V' \subseteq V \) and \( E' \subseteq E \), such that

1. \( V' \) contains global resources, but no local resources, i.e.
   \[
   R^{\text{global}} \subseteq V' \land R^{\text{local}} \cap V' = \emptyset,
   \]

2. for each component \( c \) there is exactly one budget (which requires at least one global resource) from \( c.R \) in \( V' \), i.e.
   \[
   \forall c \in C : |\{ b | b \in c.R \land b.R \cap R^{\text{global}} \neq \emptyset \land b \in V' \}| = 1,
   \]

3. budgets requiring only local resources are ignored
   \[
   \forall b \in B : b.R \subseteq R^{\text{local}} \Rightarrow b \notin V',
   \]

4. \( E' \) contains all the edges (and only those edges) from \( E \) which have both endpoints in \( V' \), i.e.
   \[
   \forall \{a, b\} \in E : (a \in V' \land b \in V') \Leftrightarrow \{a, b\} \in E'.
   \]

Only budgets which require at least one global resource will wait inside of a resource queue. Budgets which require only local resources will never be inserted into a resource queue, because resource queues are associated exclusively with global resources. Condition 1 in Definition 2.5 makes sure that budgets which require only local resources will be unreachable from global budgets. Condition 3 removes those budgets from a partial budget requirements graph to keep it concise.

Definition 2.6. We define \( \text{partial}(G) \) as the set of all possible partial budget requirements graphs which can be derived from the budget requirements graph \( G \).

Lemma 2.4. The \( \text{partial}(G) \) set contains

\[
\prod_{c \in C} |\{ b | b \in c.R \land b.R \cap R^{\text{global}} \neq \emptyset \}| \leq \prod_{c \in C} |c.R|
\]

graphs.

Proof. It follows directly from conditions 2 and 3 in Definition 2.5. \( \square \)

Figure 2.6 illustrates the partial graphs derived from the budget requirements graph in Figure 2.1.
Figure 2.6: Partial budget requirements graphs derived from the budget requirements graph in Figure 2.1, assuming components $C = \{a, b, c, d\}$, with $a.R = \langle a_1 \rangle$, $b.R = \langle b_1 \rangle$, $c.R = \langle c_1, c_2 \rangle$, $d.R = \langle d_1, d_2 \rangle$.

**Definition 2.7.** Let $G = (V,E)$ be a budget requirements graph. We define a function $\text{dependent}(b, g) : \mathcal{B} \times (2^V \times 2^E) \rightarrow 2^\mathcal{B}$ to be a set of budgets which $b$ depends on in the budget requirements graph $g$. We say that “budget $b$ depends on budget $a$” if both belong to the same connected subgraph of graph $G$.

Notice that the dependency relation is symmetric, i.e.

$$a \in \text{dependent}(b, g) \iff b \in \text{dependent}(a, g), \quad (2.6)$$

and transitive, i.e.

$$a \in \text{dependent}(b, g) \land b \in \text{dependent}(c, g) \Rightarrow a \in \text{dependent}(c, g). \quad (2.7)$$

**Example 2.2** Figure 2.7 shows an example of the dependencies in the partial budget requirements graphs in Figure 2.6.

<table>
<thead>
<tr>
<th>$b$</th>
<th>$\text{dependent}(b, G_a)$</th>
<th>$\text{dependent}(b, G_b)$</th>
<th>$\text{dependent}(b, G_c)$</th>
<th>$\text{dependent}(b, G_d)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$e_1$</td>
<td>${d_1, e_1, f_1}$</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>$e_2$</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
<td>${d_1, e_1, f_1}$</td>
<td>${e_1, f_1}$</td>
</tr>
<tr>
<td>$d_1$</td>
<td>${c_1, e_1, f_1}$</td>
<td>$\emptyset$</td>
<td>${c_2, e_1, f_1}$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>$d_2$</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>$e_1$</td>
<td>${e_1, d_1, e_1}$</td>
<td>$f_1$</td>
<td>${c_2, d_1, f_1}$</td>
<td>${c_2, f_1}$</td>
</tr>
<tr>
<td>$f_1$</td>
<td>${c_1, d_1, e_1}$</td>
<td>${e_1}$</td>
<td>${c_2, d_1, e_1}$</td>
<td>${c_2, e_1}$</td>
</tr>
</tbody>
</table>

Figure 2.7: Budget dependencies for global budgets in Figure 2.6, where $G_a, G_b, G_c, G_d$ represents the partial budget requirements graph in Figure 2.6.a, 2.6.b, 2.6.c, 2.6.d, respectively.
Lemma 2.5. Under PSRP, each budget $b \in B$ will have to wait before it can start executing for at most

$$\text{wait}(b) = \max_{g \in \text{partial}(G)} \sum_{k \in \text{dependent}(b, g)} k.E,$$

(2.8)

where $G$ is the budget requirements graph.

Proof. Consider the situation when a budget $b$ tries to start executing and acquire resources in $b.R$. If any of the resources is not available, $b$ will have to wait. Let $\text{wait}_\text{resource}(b, r)$ be the worst-case time that budget $b$ may spend waiting due to resource $r$. A resource $r$ can be either local or global.

If $r$ is a local resource, then (according to Definition 2.4) $b$ must be a local budget. In this case, (according to Lemma 2.3) $b$ is guaranteed to require a preemptive resource $p$. It will therefore be started only when its priority is higher than the preemption threshold of $p$, i.e. when (according to Rule 1) all the nonpreemptive resources required by $b$ are available. Hence, $b$ will never need to wait on $r$, i.e. $\text{wait}_\text{resource}(b, r) = 0$.

If $r$ is a global resource, then $\gamma(b)$ will be added to the resource queue $r.queue$. Since $r.queue$ is a FCFS queue, a budget $k$ waiting inside of $r.queue$ in front of $b$ will have to complete first, before $\gamma(k)$ can be added at the end of $r.queue$ again.

Hence a component may be represented only once inside of a resource queue, and therefore the length of the resource queue is at most equal to the number of components requiring $r$. In other words, a component $c$ sharing resource $r$ with budget $b$ will interfere with $b$ (during the time $b$ is waiting on $r$) for the duration of at most one of its budgets in $c.R$.

Let $B(b, r)$ be the worst-case set of budgets which are waiting in $r.queue$ in front of $b$. Each budget $k \in B(b, r)$ can itself be waiting on other resources: for each resource $s \in k.R$, budget $k$ may need to wait for all budgets in $B(k, s)$, for each of those budgets in $B(k, s)$ we can continue the same reasoning. In effect, budget $b$ may need to wait for many budgets which it indirectly depends on. A straightforward approach would be to designate all budgets which are reachable from $b$ in $G$ as the set that budget $b$ depends on. We now show how to bound this set by removing the unnecessary vertices from $G$.

(i) The fact that $k$ is inside of a resource queue implies that its priority is higher or equal to the preemption threshold of any preemptive resource it may require, meaning that it cannot be waiting for local resources. We therefore need to consider only global resources.

(ii) At any moment in time only one budget of a component can be active. Therefore, budget $b$ will not depend on budgets belonging to the same component, i.e budgets in $\gamma(b).R \setminus \{b\}$.

(iii) Moreover, budget $b$ is will not depend on any budget which a budget from the same component depends on, unless $b$ also depends on it after removing $\gamma(b).R \setminus \{b\}$ from $G$. The same holds for any other budget.

According to (i), (ii) and (iii) we need to consider only budgets which are reachable from $b$ in $G$, after we remove the vertices corresponding to the local resources, and budgets belonging to the same component from $G$. In other words, budget $b$ depends only on budgets $k$, such that (according to Definition 2.6 and 2.7) $k \in \text{dependent}(r, g)$, where $g \in \text{partial}(G)$. Moreover, since (according to
2.1) there are no dependency cycles, we need to consider only a single instance of each $k$.

Budget $b$ will have to wait for $\text{wait}_{\text{resource}}(b, r)$ time on all resources $r \in b.R$. Since a budget is inserted into the resource queues of all resources $r \in b.R$ simultaneously, and any spin locks are performed concurrently, its total waiting time is given by (2.8).

Corollary 2.6. A budget $b$ which requires only local resources will never have to wait inside of a resource queue, i.e.

$$\forall b \in B : b.R \subseteq R^{\text{local}} \Rightarrow \text{wait}(b) = 0$$

Definition 2.8. For budget $b$ we use $b.E' = \text{wait}(b) + b.E$ to denote the execution time of $b$ extended with its waiting time.

Definition 2.9. We define $A : B \rightarrow T$ as a function, where $A(b) = t$ means that budget $b$ has an activation time $t$ relative to the release time of its parent component. $A(b)$ is equal to the completion time of $\text{prev}(b)$, or 0 in case $b$ is the first budget, i.e.

$$A(b) = \begin{cases} \text{WR}(\text{prev}(b)) & \text{if index}(b) > 1, \\ 0 & \text{otherwise}. \end{cases} \quad (2.9)$$

Theorem 2.7. Under PSRP, the worst-case response time of a global budget $b \in B^{\text{global}}$, measured since the arrival of the parent component, is bounded by

$$\text{WR}(b) = A(b) + b.E' \quad (2.10)$$

Proof. Since each budget belonging to component $c$ is dispatched only after the previous budget in $c.R$ has completed (or when $c$ has arrived, in case of the first budget), and since we assumed $c.D \leq c.T$ for all components $c$, budgets belonging to the same component do not interfere with each other. Since each budget is dispatched immediately after the previous one has completed (or at the moment $c$ has arrived, in case of the first budget), there is no idle time between the budgets. Therefore, budget $b$ will attempt to lock its required resources at time $A(b)$.

At this moment it will start waiting on all the resources which it requires but which are unavailable. It will wait for at most $\text{wait}(b)$ time units.

Since budgets spin at the highest priority, immediately after it stops spinning it will start executing. Also, since we assumed that all nested critical sections have been shifted outwards and since the preemption threshold of all resources in $b.R$ is raised to $\pi_+$ at the moment $b$ starts executing, budget $b$ cannot be pre-empted nor blocked once it starts executing. In the worst-case it will therefore execute for $b.E'$ time before completing. In order to compute the worst-case response time of a global budget $b$, we therefore simply have to sum up its release jitter, total waiting time and execution time.

Response time of local budgets

In this section we derive the worst-case response time of a local budget.
Lemma 2.8. Under PSRP, the maximum blocking that a local budget $b$ can experience is given by

$$B(b) = \begin{cases} \max\{B_{\text{local}}(b), B_{\text{global}}(b)\} & \text{if } \forall r \in b.R : r \in \mathcal{R}_{\text{local}}, \\ B_{\text{local}}(b) & \text{otherwise.} \end{cases}$$

(2.11)

where

$$B_{\text{local}}(b) = \max \{ k.E' \mid k.R \cap \mathcal{R}_{\text{global}} = \emptyset \land k.\pi > b.\pi \land (\exists r \in k.R \cap b.R : \varphi(r) \leq b.\pi) \}$$

(2.12)

and

$$B_{\text{global}}(b) = \max \{ k.E' \mid k.R \cap \mathcal{R}_{\text{global}} \neq \emptyset \land k.\pi > b.\pi \land k.R \cap b.R \neq \emptyset \}$$

(2.13)

Proof. A local budget $b$ can be blocked by local and global budgets. Let $B_{\text{local}}(b)$ and $B_{\text{global}}(b)$ be the blocking time experienced by $b$ due to local and global resources, respectively.

Global budgets use only global resources (according to Definition 2.4). Local budgets therefore only compete with local budgets on local resources. Access to local resources is managed using SRP. According to SRP, budget $b$ may be blocked by a lower priority budget only once, before $b$ starts executing. Moreover, this blocking time is equal to the length of the longest budget among those which have a lower priority than $b$, and share resources with $b$ which have a resource ceiling greater than or equal to the priority of $b$. Equation (2.12) follows.

A local budget $b$ which requires only local resources may also be blocked by a lower priority local budget $k$ which requires also global resources, when it spinlocks or executes on those global resources. According to Lemma 2.3, every local budget uses exactly one local preemptive resource. The PSRP algorithm allows a budget to execute the $\text{lock}()$ operation only if its priority is higher than the preemption threshold of the local preemptive resource shared with $b$. Since the ready budgets are scheduled on the preemptive resource according to their priority, $b$ can be blocked by only one budget $k$ and at most once. Moreover, $k$ must have started executing before $b$ has arrived, otherwise $b$ would have been scheduled instead. According to Lemma 2.7 the resource holding time of budget $k$ on each of its required resources is bounded by $k.E'$. Equation (2.13) follows.

Since (according to Definitions 2.3 and 2.4) exactly one preemptive resource will be shared between all local budgets sharing resources with $b$, this preemptive resource will synchronize the access to all other (non-preemptive) resources required by $b$. Budget $b$ which requires only local resources can therefore block on either a local budget or a global budget, but not both. The first condition in Equation (2.11) follows.

A local budget $b$, which requires at least one global resource, will start spinning at the highest priority as soon as it reaches the highest priority on the preemptive resource. Since the spinning time is already taken into account in $b.E'$, we only need to consider blocking on local budgets, and can ignore blocking on global budgets. The second condition in Equation (2.11) follows.

Example 2.3 Applying Lemma 2.8 to our leading example in Figure 2.1 (with budget priorities decreasing alphabetically) will result in the following blocking times for local budgets:

$$B(a_1) = \max\{b_1.E', c_1.E'\}, \quad B(b_1) = c_1.E', \quad B(c_1) = 0.$$
Notice that Lemma 2.8 ignores the fact that \( c_1 \) may block on \( d_1 \), since it is taken into account in the \( c_1.E' \) term in Theorem 2.9.

**Theorem 2.9.** Under PSRP, the worst-case response time of a local budget \( b \in \mathcal{B}^{\text{local}} \), measured since the arrival of the parent component, is bounded by

\[
WR(b) = A(b) + w(b)
\]

where \( w(b) \) is the smallest value which satisfies

\[
w(b) = B(b) + b.E' + \sum_{k | k \leq b, k \notin \{k : R \cap b.R \neq \emptyset \}} \left\lceil \frac{w(b) + A(k)}{k.T} \right\rceil k.E'
\]

**Proof.** As soon as a local budget is released, it will try to lock all its required resources in \( b.R \). If any of the resources it requires are not available, it will block for \( B(b) \) given by (2.11). When \( b \) is ready to resume after the initial blocking, we distinguish between two cases, depending on whether (i) \( b \) requires only local resources, or (ii) \( b \) requires at least one global resource.

In case (i), according to Corollary 2.6, budget \( b \) will not wait inside of a resource queue, i.e. \( \text{wait}(b) = 0 \). During the time that the budget is blocked or executing, higher priority budgets sharing resources with \( b \) can arrive and interfere with it. These budgets must be local too, otherwise, according to (2.3), budget \( b \) would have been global. The interarrival time between two consecutive invocations of a higher priority budget \( k \) is equal to its components period, with the first arrival suffering a release jitter \( A(k) \). Equation (2.15) follows.

In case (ii), during the time \( b \) is blocked, higher priority budgets may arrive. However, since \( b \) requires a global resource, as soon as it becomes ready to execute it will be inserted into the resource queue of all resources in \( b.R \) and start spinning at the highest priority on the single local preemptive resource which it requires (according to Lemma 2.3). The spinning time is included in the \( b.E' \) term in (2.15). As soon as all the resources in \( b.R \) are available, it will continue executing at the highest priority on the preemptive resource. Therefore, higher priority budgets arriving during the time \( b \) is waiting or executing (i.e. “during” the \( b.E' \) term) will not interfere with \( b \). Since in this theorem we are providing an upper bound, equation (2.15) follows.

A local budget \( b \) will be delayed (relative to the arrival of its parent component) by the worst-case response time of the previous budget (if any), represented by the \( A(b) \) term in (2.14). 

**Response time of components**

Now that we know how to compute the worst-case response time of local and global budgets, we can easily determine the worst-case response time of components.

**Corollary 2.10.** Under PSRP, the worst-case response time of a component \( c \in C \) is given by the worst-case response time of the last budget in \( c.R \)

Notice that this analysis resembles the holistic scheduling analysis presented by Tindell and Clark (1994). They describe the end-to-end delay for a pipeline of tasks in a distributed system, where each task is bound to a processor and can
trigger a task on another processor by sending a message of a shared network. Their tasks correspond to our budgets, and their pipelines of tasks correspond to our components. However, in their model each task executes on a single processor and may require only local nonpreemptive resources.

**Example 2.4** Earlier in this chapter we have observed that a common approach for scheduling tasks on a platform comprised of multiple heterogeneous resources is to treat the whole platform as a whole, allowing at most one task to use the platform at a time. In this example we compare this approach to PSRP and show that PSRP can indeed exploit the concurrency available on a multi-resource platform by verifying for an example task set with a utilization greater than 1 all deadlines are met.

Consider our leading example platform from Figure 2.1, comprised of three processors \( P = \{p_1, p_2, p_3\} \), three logical resources \( N = \{n_1, n_2, n_3, n_4\} \), executing an application consisting of components \( C = \{a, b, c, d, e, f\} \) and budgets \( B = \{a_1, b_1, c_1, d_1, d_2\} \), specified in Figure 2.8. The corresponding partial budget requirements graphs are shown in Figure 2.6 with the derived dependencies for global budgets in Figure 2.7.

<table>
<thead>
<tr>
<th>Component</th>
<th>( \pi )</th>
<th>( O )</th>
<th>( T )</th>
<th>( D )</th>
<th>( R )</th>
<th>Budget</th>
<th>( E )</th>
<th>( R )</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>1</td>
<td>0</td>
<td>14</td>
<td>14</td>
<td>( a_1 )</td>
<td>( a_1 )</td>
<td>b_1</td>
<td>{ p_1, n_1 }</td>
</tr>
<tr>
<td>b</td>
<td>2</td>
<td>0</td>
<td>14</td>
<td>14</td>
<td>( b_1 )</td>
<td>c_1</td>
<td>3</td>
<td>{ p_1, n_1, n_2 }</td>
</tr>
<tr>
<td>c</td>
<td>3</td>
<td>0</td>
<td>14</td>
<td>14</td>
<td>( c_1, c_2 )</td>
<td>c_2</td>
<td>1</td>
<td>{ p_2, p_3, n_2 }</td>
</tr>
<tr>
<td>d</td>
<td>4</td>
<td>0</td>
<td>14</td>
<td>14</td>
<td>( d_1, d_2 )</td>
<td>d_1</td>
<td>1</td>
<td>{ p_3, n_2, n_3 }</td>
</tr>
<tr>
<td>e</td>
<td>5</td>
<td>0</td>
<td>14</td>
<td>14</td>
<td>( e_1 )</td>
<td>d_2</td>
<td>8</td>
<td>{ n_3 }</td>
</tr>
<tr>
<td>f</td>
<td>6</td>
<td>0</td>
<td>14</td>
<td>14</td>
<td>( f_1 )</td>
<td>e_1</td>
<td>1</td>
<td>{ p_3, n_4 }</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>f_1</td>
<td>1</td>
<td>{ p_3, n_4 }</td>
</tr>
</tbody>
</table>

Figure 2.8: Component and budget specifications.

We use Lemmas 2.7 and 2.9 to compute the worst case response times of all budgets in \( B \), and hence of all the components in \( C \).

Budget \( a_1 \) will experience local blocking due to \( b_1 \) and global blocking due to \( c_1 \),

\[
\begin{align*}
B^{local}(a_1) &= b_1.E' = wait(b_1) + b_1.E \\
B^{global}(a_1) &= c_1.E' = wait(c_1) + c_1.E \\
B(a_1) &= \max\{B^{local}(a_1), B^{global}(a_1)\} = \max\{b_1.E', c_1.E'\}
\end{align*}
\]

Since \( b_1 \) requires only local resources, according to Corollary 2.6, \( wait(b_1) = 0 \).

We can compute \( wait(c_1) \) using Lemma 2.5. For each partial budget requirements graphs derived from \( G \) we need to compute the set of budgets which \( c_1 \) depends on and sum up their execution times. \( wait(c_1) \) is then equal to the maximum of these sums. According to Figure 2.7, the only dependency sets for \( c_1 \) among all partial budget requirements graphs budget \( c_1 \) is \( \{d_1, e_1, f_1\} \).

Therefore,

\[
wait(c_1) = d_1.E + e_1.E + f_1.E.
\]

and

\[
\begin{align*}
b_1.E' &= 0 + 2 = 2 \\
c_1.E' &= 1 + 1 + 1 + 3 = 6.
\end{align*}
\]
The worst-case response time of budget $a_1$ is given, according to Lemma 2.9, by

$$WR(a_1) = A(a_1) + w(a_1)$$

with

$$w(a_1) = B(a_1) + a_1.E' + \sum_{k \in \{a \neq b \land \forall \pi \in (k.R \lor b.R \neq \emptyset)\}} \left[ \frac{w(a_1) + A(k)}{k.T} \right] k.E'$$

Since $a_1$ is the first budget in the sequence $a.R$, it will suffer no jitter, i.e. $A(a_1) = 0$. Since it requires only local resources, according to Corollary 2.6, $wait(a_1) = 0$. Since $a_1$ belongs to the highest priority component, it will not be preempted while executing. Therefore,

$$WR(a_1) = \max\{b_1.E', c_1.E'\} + a_1.E = \max\{2, 6\} + 2 = 8.$$ 

Similarly, we can compute the worst-case response time for local budget $b_1$

$$B_{local}(b_1) = 0$$
$$B_{global}(b_1) = c_1.E'$$
$$B(b_1) = c_1.E'$$
$$WR(b_1) = c_1.E' + b_1.E + a_1.E = 6 + 2 + 2 = 10$$

Since the local budget $c_1$ requires a global resource $u_2$, according to (2.11) its blocking time is bounded by $B_{local}(c_1)$. Since $c_1$ has a lower priority than $a_1$ and $b_1$, it can be preempted by both of them. Therefore,

$$B_{local}(c_1) = 0$$
$$B(c_1) = 0$$
$$WR(c_1) = c_1.E' + a_1.E + b_1.E = 6 + 2 + 2 = 10$$

Budget $c_2$ is a global budget. Its worst-case response time is therefore given by Lemma 2.7. According to Figure 2.7, the dependency sets for $c_2$ are $\{d_1, e_1, f_1\}$ and $\{e_1, f_1\}$. Therefore,

$$wait(c_2) = \max\{d_1.E + e_1.E + f_1.E, e_1.E + f_1.E\} = 1 + 1 + 1 = 3$$
$$c_2.E' = wait(c_2) + c_2.E = 3 + 1 = 4$$
$$WR(c_2) = WR(c_1) + c_2.E' = 10 + 4 = 14$$

Budget $d_1$ is a global budget. According to the dependencies in Figure 2.7 we get

$$wait(d_1) = \max\{c_1.E + e_1.E + f_1.E, c_2.E + e_1.E + f_1.E\}$$
$$= \max\{3 + 1 + 1, 1 + 1 + 1\} = 5$$
$$WR(d_1) = wait(d_1) + d_1.E = 5 + 1 = 6$$

Budget $d_2$ is a global budget. According to Figure 2.7 it does not depend on any other budgets. Therefore,

$$wait(d_2) = 0$$
$$WR(d_2) = WR(d_1) + wait(d_2) + d_2.E = 6 + 8 = 14$$

According to Corollary 2.10, the worst-case response times of components $a, b, c, d$ are 6, 8, 10, 10, respectively. Since the worst-case response time of each component is smaller or equal to its deadline, the system is schedulable. The utilization of the system is

$$U(a) + U(b) + U(c) + U(d) + U(e) + U(f) = \frac{2}{14} + \frac{2}{14} + \frac{3 + 1}{14} + \frac{1 + 8}{14} + \frac{1}{14} + \frac{1}{14} = \frac{19}{14}$$
which is greater than 1, so we are exploiting some of the available concurrency.

Time complexity of the analysis

The worst-case response time of all components can be computed using the following procedure:

1. Construct the budget requirements graph $G$.
2. Compute the partial budget requirements graphs $\text{partial}(G)$.
3. For each partial budget requirements graph compute its connected components.
4. For each global budget $b \in B^{\text{global}}$ compute its waiting time $\text{wait}(b)$.
5. For each component $c \in C$, starting with the highest priority one, traverse its required budgets sequence $c.R$, starting with the first one, and compute its worst-case response time.

Lemma 2.11. The worst-case response time of all components in $C$ can be computed in pseudo-polynomial time.

Proof. The time complexity is given by the sum of the complexities of the individual steps of the above procedure.

Step 1: A budget requirements graph $G = (V, E)$ is a tripartite graph which contains $|V| = |R| + |B|$ vertices and $|E| = |I|$ edges, where $I = \bigcup_{b \in B} b.R$ is the set of all budget items. It can be constructed in $O(|R| + |B| + |I|)$ time.

Step 2: The set of all partial budget requirements graphs $\text{partial}(G)$ contains $|\text{partial}(G)| = \prod_{c \in C} |c.R|$ graphs. Each partial graph can be computed by removing budget-nodes and connecting edges from the original graph $G$, which can be done in $O(|R| + |B| + |I|)$ time.

Step 3: Each partial graph can be decomposed into connected components by performing a breadth first search in $O(|R| + |B| + |I|)$ time.

Step 4: By precomputing the sum of the execution times of each connected component in all partial budget requirements graphs (which can be done while constructing the connected components in Step 3), we can can compute the $\sum_{k \in \text{dependent}(b,g)} k.E$ term in (2.8) in $O(1)$ time, by subtracting its execution time from the sum for each partial budget requirements graph. This has to be repeated for each partial budget requirements graph to compute the max term in (2.8).

Step 5: The worst-case response time of a local budget in (2.14) relies on solving the recursive equation (2.15), which (according to Lehoczky et al. (1989)) can be done for all local budgets in pseudo-polynomial time. Steps 1-4 show that the worst-case response time for all global budgets can be computed in polynomial time, while step 5 shows that for all local budgets it can be done in pseudo-polynomial time. Therefore, using Corollary 2.10, the worst-case response time for all components can be computed in pseudo-polynomial time.
2.4.4 Discussion

Pessimistic analysis for local budgets Lemma 2.9 describes the worst-case response time of a local budget. It treats all local budgets alike, whether they require global resources or not. However, only a local budget which requires *only* local resources can be preempted by higher priority budgets while it is executing. A budget which requires at least one global resource will be scheduled non-preemptively on all preemptive resources it requires. We can therefore lower the bound on worst-case response time of local budgets which require global resources by ignoring the interference of higher priority tasks during the budget execution. For this purpose we can apply the schedulability analysis for Fixed-Priority with Deferred Preemption Scheduling (FPDS) by Bril et al. (2007).

Nested global critical sections Nested critical sections can lead to deadlock. MSRP explicitly forbids nested global critical sections. FMLP supports nested critical sections by means of resource groups. The resource groups partition the set of resources into independent subsets. Consequently, a task trying to access resource $r$ may become blocked on all resources in the resource group $G(r)$.

Under PSRP, if a task has nested critical sections we can move the inner critical sections outwards until they overlap exactly with the outer most critical section. This new task can be expressed in our system model, where budgets require several resources at the same time. Each budget can be blocked only on resources which it requires (rather than the complete resource group). PSRP therefore provides a more flexible approach for dealing with nested global critical sections than FMLP. In the worst case, however, under PSRP a task will be indirectly blocked by all dependent budgets, while under FMLP it will be blocked by ...

Under FMLP every time a job is resumed it may block on a local resource. This is the same for PSRP, where we have to include the blocking time for each budget (rather once per component).

Multiprocessors without migrating tasks Since each budget of a component $c$ can start on a different set of processors from the previous budget in $c.R$, our model can express migrating components on multiprocessors, where components migrate on budget boundaries. Since every budget may execute on a different processor, in (2.9) we need to take the blocking time $B(b)$ into account for *every* budget in $c.R$. However, if we limit the model to non-migrating components where all budgets belonging to the same component require a fixed set of processors, like the MSRP algorithm does, then only the first budget in $c.R$ may block. In that case we must include the blocking term *only* for the first budget, and can ignore it for all following budgets.
Bibliography


2.4 Bibliography


