1 Introduction

Since the introduction of affordable multi core processors, concurrency becomes more important than ever. The introduction of technologies like Google apps and Microsoft Azure also increases the need to understand and use the full potential of concurrency. Applications that offer concurrent solutions are necessary; when an application wants to use the full potential of a multi core processor but are far more complex than their single threaded equivalent.

Research like thread algebra provides a way to acquire an understanding of some essential aspects of multi threaded programs. By translating real programs to a more general and fundamental theory, the result is a more abstract view and description of a programs behavior. In this paper we propose and implement an extension onto the current strategies as described in [2] which provides us the usage of multiple prioritized queues.

The motivation for this extension is to bring current strategies to a level where they can be used for reasoning about kernel behaviors which are used in operating systems. The newly created strategies are not mend to be a full description of modeling a windows kernel, because it lacks some important features such as assigning a higher priority to a thread or thread termination. The new extensions must be seen as an initial attempt to make current strategies of thread algebra more compatible with kernel scheduling.

Before we start with the proposed extensions we begin with a short introduction about the windows scheduler. It gives a good impression about what we want to achieve and it is a useful example which we can refer to. The next chapter gives an introduction into thread algebra and cyclic strategic interleaving which is described in [2]. After this chapter the next three chapters are about the new extensions. (i) Extending cyclic strategic interleaving with one other queue using non-blocking actions. (ii) Extending cyclic strategic interleaving with multiple queues using non-blocking actions. (iii) Extending cyclic strategic interleaving with multiple queues using blocking actions. We combine
2 Introduction to scheduling threads on Windows

Current interleaving strategies described in [2] are too basic to describe the behavior of threads in the windows scheduler or other operating systems using multiple queues. To make it more close to practical purposes we want to expand Thread Algebra to model the scheduler that is used in Windows. Before we start with extending current thread algebra we will first give an introduction how the windows scheduler works.

The windows scheduler uses a scheduling algorithm called multilevel feedback queues. The processes are divided into multiple queues and each queue has its own priority. The thread that is running is always the process with the highest priority that is available. With Thread Algebra you might think that some processes never get any processor time, but in most real-life cases processes are blocked, waiting on some event, for example a time interval, user input or disk I/O.

The windows scheduler uses 32 queues (0-31) where 31 is the highest priority and 0 the lowest priority (figure 1). The queues 16 - 31 are used for real-time processes and 1-15 are variable priority classes, priority 0 is reserved for memory.

Figure 1: Windows scheduler overview
management thread (figure 2). Each queue contains a linked list with processes which are assigned to this queue.

The processes in the variable class of processes starts with a base priority which is based on the thread priority and the process priority, but the windows executive modifies the priorities during run-time. By modifying the priorities at runtime it can give a higher priority to interactive processes. So when a user presses a button, the scheduler gives priority to the process that receives the button event and gets time to respond to that event. The user gets a feeling that the process is interactive. The variable process priorities can not fall below the lower range of the thread base priority and cannot exceed 15. When a process used up its entire time quantum its priority might drop. When it is waiting for disk I/O it receives a small boost but when it waits for a keyboard event it receives a large boost. Thus, interactive threads tend to have the highest priority within the variable priority class.

The windows scheduler modifies its priorities, however, with thread algebra we leave this part out of scope. As it requires a more advanced scheduling algorithm than a standard Multi level feedback queue. Also it is described that the priorities get a boost but it is not described exactly how large that boost is. The expansion of modifying priorities might be a topic for further research.

The time quantum that a thread receives varies for different versions of Windows. Before Windows Vista, a thread received a time quantum to use for its action. However, hardware and software interruptions also made use of the processor time. It was possible that those interruptions consumed most of its time quantum before the scheduler interrupted and scheduled a new thread [1]. Windows Vista and newer versions use the cycle counter register of modern register to keep track of how many cycles a thread executes. The Windows Vista scheduler does not count interruptions against a thread’s turn, this turns out in a greater fairness in program execution and a more deterministic application behavior.

3 Introduction to thread algebra

Current research [2] outlines an algebraic theory about threads and multi-threading using cyclic interleaving as their base interleaving strategy. In addition they combine this strategy using additional features like (non-)blocking actions, thread creation, thread termination and step counting or combine two or more features.
Thread algebra is an extension of Basic Polarized Process Algebra (BPPA) [3]. Just like BPPA, thread algebra focuses on the semantics of deterministic sequential programs but thread algebra extends this with a deterministic interleaving strategy which determines how threads are interleaved. The most basic deterministic interleaving strategy is cyclic interleaving which takes the first basic action from a thread vector and the remaining thread vector undergoes cyclic permutation. By this we mean that the first thread becomes the last thread and all others move one position to the left.

BPPA and thread algebra use a fixed but arbitrary finite set \( A \) of basic actions with \( \tau \in A \) and has the following constants and operators:

- the deadlock constant \( D \);
- the termination constant \( S \);
- for each \( a \in A \), a binary postconditional composition operator \( _- \preceq a \succeq _- \).

A post conditional composition operator \( _- \preceq a \succeq _- \) is abbreviated as: \( a \circ p \), and is called action prefixing. Each basic action is taken as a command to be processed by the execution environment. At completion of each action it yields a reply value which is T or F. In short we call this kind of actions non-blocking actions. Blocking actions are basic actions which can be processed by a defined set of services \( f \in (F_{pts}, F_{pils}, F_{tss}) \). A service can be in a state where not all commands are enabled which leads to blocking actions. The action \( \tau \) always yields true and can be written as follows: \( x \preceq \tau \succeq y = \tau \circ x \).

As described in this paper current operating systems like Windows XP and Windows 7 use a scheduling algorithm which is based on multiple vectors containing thread vectors. The goal of this paper is to extend thread algebra for strategic interleaving described in [2] with multiple vectors containing thread vectors which we will call multiple queues. We also introduce an extension for thread creation, step counting and a combination of those two features for both blocking and non-blocking actions.

3.1 Blocking actions with step counting, forking and memory

When reviewing paper [2] we found some issues in table 24. This table contains axioms which define the behavior of threads using a combination of the features step counting, forking and memory. We detected one minor shortcoming, when the enableness test returns the current step as a result the next thread runs less than \( k \) steps. We demonstrate this in the next example.
When using a fixed set of priorities we can choose to tag the threads independently. In the above description we used the notion of "CPU class with the highest priority is routed before other traffic.

Another example is the implementation scheduling of Windows which creates one long vector which holds all threads. The second one creates a fixed interleaving strategy between the actions of the same priority level. The implementation of this can be done most easily using a strategy which identifies an interleaving strategy between the actions of the same priority level. The Windows scheduler determines which thread needs some CPU until threads with a higher priority are finished or stalled.

A second example is called "Multi-Level Priority Queues" and is implemented in enterprise Cisco routers [4]. Every traffic type can be assigned to some traffic class which is tagged with some priority level. Traffic which is assigned to the class with the highest priority is routed before other traffic.

Current thread algebra has no means to provide a solution for modeling multiple queues. Our goal is to extend current theory with an extension to provide this kind of modeling. In above description we used the notion of "CPU

\[ f.lock \circ a \circ c \circ f.unlock \circ b \circ S \]
Table 1: Examples for cyclic interleaving using two queues
\[
\begin{align*}
\parallel \text{csi-2pq} (\langle X \rangle \hookrightarrow \langle Y \rangle)^1 (\langle W \rangle \hookrightarrow \langle Z \rangle)^2 \\
\parallel \text{csi-2pq} (\langle X \rangle \hookrightarrow \alpha)^1 (\langle B \rangle)^2
\end{align*}
\]
time” instead of this we perform one or more actions without considering the time a single action takes to complete.

3.3 Example
By using an example we demonstrate how we extend the notation of thread algebra with multiple queues. We introduce \( p \) as a parameter which represents the total queues. Every queue is annotated with a number which represents its priority. Queue number one has highest priority and queue number \( n \) the most lowest. The \( B \) represent a finite sequence of queues.

4 Two queues with non-blocking actions
This paragraph is about our first attempt to extend existing axioms described in [2] with more than one queue. To determine if our proposed extension is possible we use the most basic imaginable, which is extending table 7 from [2] with one additional queue. Where \( a \) stands for an arbitrary action from \( (A)_{tau} \).

The process and resulting axioms will be briefly discussed. We also implement other strategies which are ‘step counting’ and ‘thread creation’ and combine them. Because of time constrains we do not implement all strategies from [2] such as thread creation and bandwidth.

In this section we annotate the strategic interleaving operator with the subscript \( 2pq (\parallel_{2pq}) \), which stands for two prioritized queues, to show that we use cyclic interleaving with two queues. After the strategic interleaving operator the queues are described which have the following signature \( (\alpha)^1 (\langle B \rangle)^2 \). The first queue is identified by the number 1 and the second queue by number 2. Actions which are located in the collection of threads in queue 2 can only be executed when queue 1 is empty \( (\langle \rangle)^1 \). The queues are interleaved using round robin. By this we mean that the first thread becomes the last thread and all others move one position to the left. We use \( \alpha \) and \( (B) \) to implicitly describe zero or more threads.

4.1 Two queues
The axioms for cyclic interleaving with two queues are described in table 3. The main reason for starting with only two queues is to create a simplified setting where we have overview and some base to continue with. This is important because our goal is to extend this with multiple strategies and combinations of them and to create an implementation with more than two queues.

When creating the axioms for cyclic interleaving using two queues we had to decide if we would delete the first queue when it becomes empty, \( (\alpha)^1 = (\langle \rangle) \). We thought it would be more convenient to have the empty thread-vector explicitly appeared because of readability. The same reason is used for numbering the
Table 2: Axioms for deadlock at termination

<table>
<thead>
<tr>
<th>Expression</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S_D(S) = D$</td>
<td>S2D1</td>
</tr>
<tr>
<td>$S_D(D) = D$</td>
<td>S2D2</td>
</tr>
<tr>
<td>$S_D(\langle x \preceq a \triangleright y \rangle) = S_D(x) \preceq a \triangleright S_D(y)$</td>
<td>S2D3</td>
</tr>
</tbody>
</table>

Table 3: Axioms for cyclic interleaving with two queues

<table>
<thead>
<tr>
<th>Expression</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ \parallel_{2pq}(\langle \rangle)^2 = S ]</td>
<td>2PQ0</td>
</tr>
<tr>
<td>[ \parallel_{2pq}(\langle S \rangle \triangleleft \alpha)^2 = \parallel_{2pq}(\langle \alpha \rangle^1(B)^2 ]</td>
<td>2PQ1</td>
</tr>
<tr>
<td>[ \parallel_{2pq}(\langle D \rangle \triangleleft \alpha)^2 = \parallel_{2pq}(\langle \alpha \rangle^1(B)^2 ]</td>
<td>2PQ2</td>
</tr>
<tr>
<td>[ \parallel_{2pq}(\langle x \preceq a \triangleright y \rangle \triangleleft \alpha)^2 = \parallel_{2pq}(\langle \alpha \triangleright \langle x \rangle \rangle^2 \preceq a \triangleright \parallel_{2pq}(\langle \alpha \triangleright \langle y \rangle \rangle^2 ]</td>
<td>2PQ3</td>
</tr>
<tr>
<td>[ \parallel_{2pq}(\langle x \preceq a \triangleright y \rangle \triangleleft \alpha)^2 = \parallel_{2pq}(\langle \alpha \triangleright \langle x \rangle \rangle^2 \preceq a \triangleright \parallel_{2pq}(\langle \alpha \triangleright \langle y \rangle \rangle^2 ]</td>
<td>2PQ4</td>
</tr>
<tr>
<td>[ \parallel_{2pq}(\langle x \preceq a \triangleright y \rangle \triangleleft \alpha)^2 = \parallel_{2pq}(\langle \alpha \triangleright \langle x \rangle \rangle^2 \preceq a \triangleright \parallel_{2pq}(\langle \alpha \triangleright \langle y \rangle \rangle^2 ]</td>
<td>2PQ5</td>
</tr>
<tr>
<td>[ \parallel_{2pq}(\langle x \preceq a \triangleright y \rangle \triangleleft \alpha)^2 = \parallel_{2pq}(\langle \alpha \triangleright \langle x \rangle \rangle^2 \preceq a \triangleright \parallel_{2pq}(\langle \alpha \triangleright \langle y \rangle \rangle^2 ]</td>
<td>2PQ6</td>
</tr>
</tbody>
</table>

Most notable is the axiom count of this extension because every type in \((A)\) is described explicitly for queue one and two. We certainly should consider another notation when using more than two threads but for two threads it does the job.

4.2 Step counting

We can extend the current axioms in table 3 to include step counting \((\parallel_{k,l}(\ldots), \ldots)\) as described in [2]. This gives every thread a fixed number \(k\) of consecutive turns. The counter \(l\) indicates the number of steps which are already performed \((1 \leq l \leq k)\). The new axioms for cyclic interleaving with two queues and step counting are given in table 4.

The first axiom 2PQs0 gives a description how to initialize step counting. To incorporate step counting we had to focus on extending 2PQ5 and 2PQ6 with step counting. When performing an action and \(l \not\leq k\) the counter \(l\) is increased otherwise \(l = k\) and context switch takes place.

4.3 Thread creation

Forking or thread creation is about forking off a new thread with some priority. The creation of a new thread can fail or succeed. For thread creation we introduce one new operator NT which has two signatures. The first is \(NT(\alpha)\) where a new thread \(\alpha\) is created in the same context and in the same queue as the parent thread (see axiom 2PQf7 and 2PQf8). The second is \(NT(\alpha, i)\) where a new thread \(\alpha\) is created in queue \(i\) where \((1 \leq i \leq k)\) (see axiom 2PQf9 and 2PQf10). The thread creation functions \(NT(\alpha)\) and \(NT(\alpha, i)\) have both axioms for queue 1 and queue 2.
Table 4: Axioms for cyclic interleaving using two queues and step counting

\[
\|_{2pqf}^1(\alpha)^1(B)^2 = \|_{2pqf}^1(\alpha)^1(B)^2 \quad 2PQ_{s0}
\]
\[
\|_{2pqf}^1(\alpha)^1(\alpha\cdot\alpha)^2 = \|_{2pqf}^1(\alpha)^1(\alpha\cdot\alpha)^2 \quad 2PQ_{s1}
\]
\[
\|_{2pqf}^1((S \land \alpha)^1(B)^2 = \|_{2pqf}^1((S \land \alpha)^1(B)^2 \quad 2PQ_{s2}
\]
\[
\|_{2pqf}^1((S \cup \alpha)^1(B)^2 = \|_{2pqf}^1((S \cup \alpha)^1(B)^2 \quad 2PQ_{s3}
\]
\[
\|_{2pqf}^1((D \land \alpha)^1(B)^2 = S_D(\|_{2pqf}^1((D \land \alpha)^1(B)^2) \quad 2PQ_{s4}
\]
\[
\|_{2pqf}^1((D \cup \alpha)^1(B)^2 = S_D(\|_{2pqf}^1((D \cup \alpha)^1(B)^2) \quad 2PQ_{s5}
\]
\[
\|_{2pqf}^1((x \leq \alpha \geq y) \land \alpha)^1(B)^2 = \quad <k = l>
\]
\[
\|_{2pqf}^1((x \land \alpha)^1(B)^2 \leq a \geq \|_{2pqf}^1((x \land \alpha)^1(B)^2 \quad 2PQ_{s6}
\]
\[
\|_{2pqf}^1((x \land \alpha)^1(\alpha \land \alpha)^2 = \quad <k = l>
\]
\[
\|_{2pqf}^1((x \land \alpha)^1(\alpha \land \alpha)^2 \leq a \geq \|_{2pqf}^1((x \land \alpha)^1(\alpha \land \alpha)^2 \quad 2PQ_{s7}
\]

Table 5: Axioms for cyclic interleaving using two queues and forking

\[
\|_{2pqf}^1(\alpha)^1(\alpha)^2 = S \quad 2PQ_{s0}
\]
\[
\|_{2pqf}^1((S \land \alpha)^1(B)^2 = \|_{2pqf}^1((S \land \alpha)^1(B)^2 \quad 2PQ_{s1}
\]
\[
\|_{2pqf}^1((S \cup \alpha)^1(B)^2 = \|_{2pqf}^1((S \cup \alpha)^1(B)^2 \quad 2PQ_{s2}
\]
\[
\|_{2pqf}^1((D \land \alpha)^1(B)^2 = S_D(\|_{2pqf}^1((D \land \alpha)^1(B)^2) \quad 2PQ_{s3}
\]
\[
\|_{2pqf}^1((D \cup \alpha)^1(B)^2 = S_D(\|_{2pqf}^1((D \cup \alpha)^1(B)^2) \quad 2PQ_{s4}
\]
\[
\|_{2pqf}^1((x \leq \alpha \geq y) \land \alpha)^1(B)^2 = \quad <k = l>
\]
\[
\|_{2pqf}^1((x \land \alpha)^1(B)^2 \leq a \geq \|_{2pqf}^1((x \land \alpha)^1(B)^2 \quad 2PQ_{s5}
\]
\[
\|_{2pqf}^1((x \land \alpha)^1(\alpha \land \alpha)^2 = \quad <k = l>
\]
\[
\|_{2pqf}^1((x \land \alpha)^1(\alpha \land \alpha)^2 \leq a \geq \|_{2pqf}^1((x \land \alpha)^1(\alpha \land \alpha)^2 \quad 2PQ_{s6}
\]
\[
\|_{2pqf}^1((\exists NT(x) \geq y) \land \alpha)^1(B)^2 = \quad <k = l>
\]
\[
\|_{2pqf}^1((\exists NT(x) \geq y) \land \alpha)^1(B)^2 \leq a \geq \|_{2pqf}^1((\exists NT(x) \geq y) \land \alpha)^1(B)^2 \quad 2PQ_{s7}
\]
\[
\|_{2pqf}^1((\exists NT(x) \geq y) \land \alpha)^1(\alpha \land \alpha)^2 = \quad <k = l>
\]
\[
\|_{2pqf}^1((\exists NT(x) \geq y) \land \alpha)^1(\alpha \land \alpha)^2 \leq a \geq \|_{2pqf}^1((\exists NT(x) \geq y) \land \alpha)^1(\alpha \land \alpha)^2 \quad 2PQ_{s8}
\]
\[
\|_{2pqf}^1((\exists NT(x) \geq y) \land \alpha)^1(\alpha \land \alpha)^2 = \quad <k = l>
\]
\[
\|_{2pqf}^1((\exists NT(x) \geq y) \land \alpha)^1(\alpha \land \alpha)^2 \leq a \geq \|_{2pqf}^1((\exists NT(x) \geq y) \land \alpha)^1(\alpha \land \alpha)^2 \quad 2PQ_{s9}
\]
\[
\|_{2pqf}^1((\exists NT(x) \geq y) \land \alpha)^1(\alpha \land \alpha)^2 = \quad <k = l>
\]
\[
\|_{2pqf}^1((\exists NT(x) \geq y) \land \alpha)^1(\alpha \land \alpha)^2 \leq a \geq \|_{2pqf}^1((\exists NT(x) \geq y) \land \alpha)^1(\alpha \land \alpha)^2 \quad 2PQ_{s10}
Table 6: Axioms for cyclic interleaving using two queues, step counting and thread creation

<table>
<thead>
<tr>
<th>Axiom</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(2pq(\alpha)^4(B)^2) = (2pq(\alpha)^4(B)^2)</td>
<td>2PQa0</td>
</tr>
<tr>
<td>(2pq(\alpha)^4(S)^2 = S)</td>
<td>2PQa1</td>
</tr>
<tr>
<td>(2pq(\alpha)^4((S) \land (\alpha)) = |_{2pq}^1(\alpha)^4(B)^2)</td>
<td>2PQa2</td>
</tr>
<tr>
<td>(2pq(\alpha)^4((S) \land (\alpha)^2) = |_{2pq}^1(\alpha)^4(\alpha)^2)</td>
<td>2PQa3</td>
</tr>
<tr>
<td>(2pq(\alpha)^4((D) \land (\alpha)^3) = SD(|_{2pq}^1(\alpha)^4(B)^2))</td>
<td>2PQa4</td>
</tr>
<tr>
<td>(2pq(\alpha)^4((D) \land (\alpha)^2) = SD(|_{2pq}^1(\alpha)^4(\alpha)^2))</td>
<td>2PQa5</td>
</tr>
<tr>
<td>(|_{2pq}^1(x \leq a \geq y) \land (\alpha)^4(B)^2)</td>
<td>2PQa6</td>
</tr>
<tr>
<td>(|_{2pq}^1(x \leq a \geq y \land (\alpha)^4(B)^2)</td>
<td>2PQa7</td>
</tr>
<tr>
<td>(|_{2pq}^1(x \geq a \geq y) \land (\alpha)^4(B)^2)</td>
<td>2PQa8</td>
</tr>
<tr>
<td>(|_{2pq}^1(x \geq a \geq y \land (\alpha)^4(B)^2)</td>
<td>2PQa9</td>
</tr>
</tbody>
</table>

4.4 Thread creation and step counting
This section is about the combination of the strategy step counting and forking. The axioms, which results from combining this, are described in table ?? and 7. Most notable is the size of axiom 2PQa9 which is responsible for thread creation in a particular queue.

5 Multiple queues with non-blocking actions
It starts to get more interesting when having the possibility to use more than two prioritized queues. For example 32 just as the windows scheduler. In this section we describe how we adjust the existing axioms we described in the last section and extent those axioms to our new requirements.

First we annotate the strategic interleaving operator based on cyclic interleaving operator with the subscript "pq" which stands for "Priority Queues". We also introduce a new parameter which represents the total queues and every queue has its own identifier which represents the queue priority. When a queue has no actions to perform the next queue can take its turn to perform actions. When a queue with a higher priority contains a new thread the current action is finished and the next action will be the one in the higher queue. This is exactly
5 MULTIPLE QUEUES WITH NON-BLOCKING ACTIONS

Table 7: Axioms for cyclic interleaving using two queues, step counting and thread creation cont’d

\[ \|_{2pqsf}^{k,l}(\langle x \leq NT(z, i) \geq y \rangle \sim \alpha)^{1}(\langle B \rangle)^{2} = \]
\[ (\|_{2pqsf}^{1}(\langle x \rangle \sim \langle z \rangle \sim \langle x \rangle)^{1}(\langle B \rangle)^{2} \leq NT \geq \|_{2pqsf}^{1}(\langle \alpha \sim \langle y \rangle \rangle^{1}(\langle B \rangle)^{2}) \]
\[ \langle i = 1 \rangle \triangleright \]
\[ (\|_{2pqsf}^{k,l}(\alpha \sim \langle x \rangle)^{1}(\langle B \rangle \sim \langle z \rangle)^{2} \leq NT \geq \|_{2pqsf}^{k-1}(\alpha \sim \langle y \rangle)^{1}(\langle B \rangle)^{2}) \]
\[ \langle k = l \rangle \triangleright \]
\[ (\|_{2pqsf}^{k,l}(\langle x \rangle \sim \alpha \sim \langle z \rangle)^{1}(\langle B \rangle)^{2} \leq NT \geq \|_{2pqsf}^{k+1}(\langle y \rangle \sim \alpha)^{1}(\langle B \rangle)^{2}) \]
\[ \langle i = 1 \rangle \triangleright \]
\[ (\|_{2pqsf}^{k,l}(\langle x \rangle \sim \alpha)^{1}(\langle B \rangle \sim \langle z \rangle)^{2} \leq NT \geq \|_{2pqsf}^{k,l+1}(\langle y \rangle \sim \alpha)^{1}(\langle B \rangle)^{2}) \]
\[ \text{2pqsf9} \]
\[ \|_{2pqsf}^{k,l}(\langle \rangle)^{1}(\langle x \leq NT(z, i) \geq y \rangle \sim \alpha)^{2} = \]
\[ (\|_{2pqsf}^{k,l}(\langle \rangle)^{1}(\langle \alpha \rangle \sim \langle x \rangle)^{2} \leq NT \geq \|_{2pqsf}^{1}(\langle \alpha \rangle \sim \langle y \rangle)^{1}(\langle B \rangle)^{2}) \]
\[ \langle i = 1 \rangle \triangleright \]
\[ (\|_{2pqsf}^{k,l}(\langle \rangle)^{1}(\langle \alpha \rangle \sim \langle z \rangle \sim \langle x \rangle)^{2} \leq NT \geq \|_{2pqsf}^{1}(\langle \rangle)^{1}(\langle \alpha \rangle \sim \langle y \rangle)^{2}) \]
\[ \langle k = l \rangle \triangleright \]
\[ (\|_{2pqsf}^{k,l}(\langle \rangle)^{1}(\langle x \rangle \sim \alpha)^{1}(\langle B \rangle \sim \langle z \rangle)^{2} \leq NT \geq \|_{2pqsf}^{k,l+1}(\langle \rangle)^{1}(\langle y \rangle \sim \alpha)^{2}) \]
\[ \text{2pqsf10} \]

the same as described in the last section but now with more than two queues.

In this section we introduce some new symbols. The first symbol we introduce is \( \mathcal{A} \) which we use to represent zero, one or a vector containing multiple queues \( \mathcal{A} = (\langle \rangle) \) or \( \mathcal{A} = (\langle \alpha \rangle)_{k} \) where \( 1 \leq k \leq l \). The superscript of the strategic interleaving operator contains two natural integer values, the first is the total of all queues and the second is the index of the current working queue \( (\|_{pq}^{m,n}) \). Where \( 1 \leq n \leq m \) and m is a natural integer which is \( 1 \leq m \).

In some cases we refer (by using a superscript number at the end of an axiom) to some additional requirement in the axiom tables. The following additional requirements are defined for axiom tables 8, 9, 10 and 11:

\[ ^{1} \text{if } A_m = (\langle \rangle)^{m} \]
\[ ^{2} \text{if } n < m \text{ and } A_n = (\langle \rangle)^{n} \]

We had to introduce a mechanism to go to the next queue when the working queue is empty. To make this possible we introduced axiom PQ2 to make this work. In short this axiom checks if the working thread is empty, when it is it adds 1 to the queue index counter (n) if n is not greater than m (see requirement 3).

5.1 Multiple queues

In the last section we described a new strategy which made it possible to use a strategy with two queues. One of the observation we made was the introduction of additional axioms for describing the additional queue. Now we have to deal with an unknown number of queues this means we need a pervasive notation which have enough expressiveness to describe a multiple queue notation.

We introduce a head \((\mathcal{A}_1 \ldots \mathcal{A}_{(n-1)})\) and tail \((\ldots \mathcal{A}_m)\) before the current working queue with index n \((\langle \alpha \rangle)^{n}\). As mentioned above an action in a queue
5 MULTIPLE QUEUES WITH NON-BLOCKING ACTIONS

Table 8: Axioms for cyclic interleaving using queues

| PQ0 | \[ | p \oplus \alpha | A = | | p \oplus \alpha | A | A \]
| PQ1 | \[ | p \oplus \alpha | A = S \]
| PQ2 | \[ | p \oplus \alpha | A = | | p \oplus \alpha | A | A \]
| PQ3 | \[ | p \oplus \alpha | A_1 \ldots A_{(n-1)} | (S \cap \alpha) | \ldots \ldots | A_m = | | p \oplus \alpha | A_1 \ldots A_{(n-1)} | (\alpha) | \ldots \ldots | A_m \]
| PQ4 | \[ | p \oplus \alpha | A_1 \ldots A_{(n-1)} | (x \leq a \geq y \cap \alpha) | \ldots \ldots | A_m = \]
| PQ5 | \[ | p \oplus \alpha | A_1 \ldots A_{(n-1)} | (\alpha \cap \langle x \rangle) | \ldots \ldots | A_m \leq a \geq | | p \oplus \alpha | A_1 \ldots A_{(n-1)} | (\alpha \cap \langle y \rangle) | \ldots \ldots | A_m \]

Table 9: Axioms for cyclic interleaving using queues and step counting

| PQ0 | \[ | p \oplus \alpha | A = | | p \oplus \alpha | A | A \]
| PQ1 | \[ | p \oplus \alpha | A = S \]
| PQ2 | \[ | p \oplus \alpha | A = | | p \oplus \alpha | A | A \]
| PQ3 | \[ | p \oplus \alpha | A_1 \ldots A_{(n-1)} | (S \cap \alpha) | \ldots \ldots | A_m = | | p \oplus \alpha | A_1 \ldots A_{(n-1)} | (\alpha) | \ldots \ldots | A_m \]
| PQ4 | \[ | p \oplus \alpha | A_1 \ldots A_{(n-1)} | (x \leq a \geq y \cap \alpha) | \ldots \ldots | A_m = \]
| PQ5 | \[ | p \oplus \alpha | A_1 \ldots A_{(n-1)} | (\alpha \cap \langle x \rangle) | \ldots \ldots | A_m \leq a \geq | | p \oplus \alpha | A_1 \ldots A_{(n-1)} | (\alpha \cap \langle y \rangle) | \ldots \ldots | A_m \]

can only be done when all queues in the head are empty or when the head is an empty set (if \[ \mathcal{A}^m = \emptyset \] or \[ \mathcal{A}^m = \emptyset \] ). The new axioms are described in table 8.

5.2 Step counting

In table 9 the result of combining the axioms from table table 4 and 8. The new axioms are very straightforward.

5.3 Thread creation

The axioms for thread creation are described in table 10. To make thread creation work we need a mechanism to determine if the new thread is located in a queue with a higher priority or lower priority. When the new thread needs to be created with a higher priority we reset the index to i which ensures us that the next action which is executed is taken from the new thread.

5.4 Thread creation and step counting

In this section we combine the features thread creation and step counting which are described in table 9 and 10. The new axioms an be found in table 11. Most notable is the increasing size of some axioms. The same observation we made already for table 11 in the last section.
Multiple queues with blocking actions

In the last two sections we used only non-blocking actions but reality is different. To solve this problem we extended the current strategy which is called memory. Following additional requirements are defined for axioms described in table 13.

Table 10: Axioms for cyclic interleaving using queues and thread creation

<table>
<thead>
<tr>
<th>Axiom</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(|<em>{pqf} A = |</em>{pqf}^{\bot} A)</td>
<td>PQ0</td>
</tr>
<tr>
<td>(|_{pqf} A = S)</td>
<td>PQ1</td>
</tr>
<tr>
<td>(|<em>{pqf} A = |</em>{pqf}^{n+1} A)</td>
<td>PQ2</td>
</tr>
<tr>
<td>(|<em>{pqf} A_1 \ldots A</em>{(n-1)} ( (S) \wedge \alpha \rangle^m \ldots A_m = |<em>{pqf} A_1 \ldots A</em>{(n-1)} ( (S) \wedge \alpha \rangle^m \ldots A_m = )</td>
<td>PQ3</td>
</tr>
<tr>
<td>(|<em>{pqf} A_1 \ldots A</em>{(n-1)} ( (D) \wedge \alpha \rangle^m \ldots A_m = S_S (|<em>{pqf} A_1 \ldots A</em>{(n-1)} ( (D) \wedge \alpha \rangle^m \ldots A_m = )</td>
<td>PQ4</td>
</tr>
<tr>
<td>(|<em>{pqf} A_1 \ldots A</em>{(n-1)} ( (x \leq a \geq y) \wedge \alpha \rangle^m \ldots A_m = )</td>
<td>PQ5</td>
</tr>
<tr>
<td>(|<em>{pqf} A_1 \ldots A</em>{(n-1)} ( (y \leq NT(z) \geq y) \wedge \alpha \rangle^m \ldots A_m = )</td>
<td>PQ6</td>
</tr>
<tr>
<td>(|<em>{pqf} A_1 \ldots A</em>{(n-1)} ( (x \leq NT(z) \geq y) \wedge \alpha \rangle^m \ldots A_m = )</td>
<td>PQ7</td>
</tr>
</tbody>
</table>

6 Multiple queues with blocking actions

In the last two sections we used only non-blocking actions but reality is different. When working with threads it is possible that some thread holds a mutex lock which makes it impossible for some other thread to perform further actions because it has to wait until the mutex is locked. In this paper we use the following classification for services which is taken from [2]. The first service is called the Para Target Local Service which is used for timed actions by only one thread. The next two services are bot services which can be used in a broader context, they are called Para Target Shared Service and Target Shared Service. The notation for both are \(F_{ptls}\) and \(F_{tss}\). In some cases we refer (by using a superscript number at the end of a axiom) to some additional requirement in the axiom tables. The following additional requirements are defined for axiom tables 12, 13 and 14:

1. If \(A_m = (\langle \rangle)^m\) \(A_n \Rightarrow (\langle \rangle)^n\)
2. If \(n < m\) and \(A_m = (\langle \rangle)^m\)
3. If \(i \in A_{ptls} \), \(j \in A_{ptss} \)

Using blocking action we need some mechanism which makes it possible to perform an action in an other queue until the action which is blocked is unblocked. To solve this problem we extended the current strategy which is called memory to work with multiple queues. In this last chapter we explicitly did not implement step counting and thread creation for blocking actions. The resulting axioms would be very similar as those described for blocking actions and the axioms described in table 13.
6.1 Blocking actions

The axioms in Table 12 can be used as an introduction without step counting and thread creation. It also has some shortcomings because these axioms...
In table 13 we describe the new axioms which incorporate memory with multiple queues in Figure 10.

### Table 12: Axioms for cyclic interleaving with blocking actions

<table>
<thead>
<tr>
<th>Expression</th>
<th>Axiom</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \parallel_{pqba} A = \parallel_A \alpha )</td>
<td>PQba0</td>
<td>1</td>
</tr>
<tr>
<td>( m, n \parallel_{pqba} A = S )</td>
<td>PQba1</td>
<td>2</td>
</tr>
<tr>
<td>( m, n \parallel_{pqba} A =</td>
<td></td>
<td>_{pqba} A )</td>
</tr>
<tr>
<td>( m, n \parallel_{pqba} A_1 \ldots A_{(n-1)} (</td>
<td>S \wedge \alpha</td>
<td>^n \ldots A_m = \parallel_{pqba} A_1 \ldots A_{(n-1)} (</td>
</tr>
<tr>
<td>( m, n \parallel_{pqba} A_1 \ldots A_{(n-1)} (</td>
<td>D \wedge \alpha</td>
<td>^n \ldots A_m = S D(</td>
</tr>
<tr>
<td>( m, n \parallel_{pqba} A_1 \ldots A_{(n-1)} (</td>
<td>\tau \circ x \wedge \alpha</td>
<td>^n \ldots A_m = \tau \circ \parallel_{pqba} A_1 \ldots A_{(n-1)} (</td>
</tr>
<tr>
<td>( m, n \parallel_{pqba} A_1 \ldots A_{(n-1)} (</td>
<td>x \leq f.m \geq y \wedge \alpha</td>
<td>^n \ldots A_m = \parallel_{pqba} A_1 \ldots A_{(n-1)} (</td>
</tr>
</tbody>
</table>

### Table 13: Axioms for cyclic interleaving with blocking actions and memory

<table>
<thead>
<tr>
<th>Expression</th>
<th>Axiom</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \parallel_{pqbam} A = \parallel_{pqbam} A_1 \ldots A_H )</td>
<td>PQbam0</td>
<td>1</td>
</tr>
<tr>
<td>( m, n, \beta \parallel_{pqbam} A = S )</td>
<td>PQbam1</td>
<td>2</td>
</tr>
<tr>
<td>( m, n \parallel_{pqbam} A =</td>
<td></td>
<td>_{pqbam} A )</td>
</tr>
<tr>
<td>( m, n \parallel_{pqbam} A_1 \ldots A_{(n-1)} (</td>
<td>S \wedge \alpha</td>
<td>^n \ldots A_m = \parallel_{pqbam} A_1 \ldots A_{(n-1)} (</td>
</tr>
<tr>
<td>( m, n \parallel_{pqbam} A_1 \ldots A_{(n-1)} (</td>
<td>D \wedge \alpha</td>
<td>^n \ldots A_m = S D(</td>
</tr>
<tr>
<td>( m, n \parallel_{pqbam} A_1 \ldots A_{(n-1)} (</td>
<td>\tau \circ x \wedge \alpha</td>
<td>^n \ldots A_m = \tau \circ \parallel_{pqbam} A_1 \ldots A_{(n-1)} (</td>
</tr>
<tr>
<td>( m, n \parallel_{pqbam} A_1 \ldots A_{(n-1)} (</td>
<td>x \leq f.m \geq y \wedge \alpha</td>
<td>^n \ldots A_m = \parallel_{pqbam} A_1 \ldots A_{(n-1)} (</td>
</tr>
<tr>
<td>( m, n \parallel_{pqbam} A_1 \ldots A_{(n-1)} (</td>
<td>\alpha \wedge D</td>
<td>^n \ldots A_m \right) )</td>
</tr>
<tr>
<td>( m, n \parallel_{pqbam} A_1 \ldots A_{(n-1)} (</td>
<td>x \leq f.m \geq y \wedge \alpha</td>
<td>^n \ldots A_m \right) )</td>
</tr>
</tbody>
</table>

will not detect if all threads are blocked and also the enabledness test can be performed twice or more without any state change in between.

### 6.2 Memory

We can use memory as a tool to provide a mechanism to avoid redundant tests. In table 13 we describe the new axioms which incorporate memory with multiple
queues. This strategy is described in [2] where they introduce V as a finite set of actions of the form $f.m$, which is used to memorize the outcome of the enabledness test. The superscript $B$ is a vector of bits one for each thread in the queue where index $n$ points to. The new axioms can deal with blocked actions but when all actions are blocked, deadlock will inevitable even when underlying queues contains actions. To solve this problem we provide an updated axiom in table 14 to demonstrate how to deal when all actions are blocked in one queue. When all actions are blocked and we are not in the last queue we add one to the index counter. This solution will not provide the desirable result because actions which are blocked will be executed because we did not provide a mechanism to jump back to a queue with a higher priority. We need a more pervasive solution which can jump back in the queue when an action in a lower queue is executed. The solution for this problem is to reset the index counter (n) to 1 and is described in table 15. This solution only deals with functions of type PTSS and we should consider to create a mechanism to provide a more efficient solution for TSS.
7 Conclusions

We have outlined an extension to thread algebra based on several existing strategic interleaving strategies. This extension involved both blocking and non-blocking actions with an possible extension of the following strategies: step counting, thread creation, memory or even a combination of them. Our objective was create a set of axioms which demonstrates if we could extend cyclic strategic interleaving to make it compatible with a scheme which involved prioritized vectors containing thread vectors. A vector containing a vector of threads we call this a queue and has it’s own priority.

This paper started with an extension which used only two queues. This result was a fairly simple notation but the drawback of this notation was the notable increase in the number of resulting axioms. Another problem raised when combining our extension with step counting and thread creation. The resulting axiom of creation a new thread using function NT(α,i) takes a lot of space which makes the axiom more complex. Unfortunately this was something we could not solve. When dealing with multiple queues we had to find a pervasive solution to deal with multiple queues. We could not extend current axioms but we had to came up with a more flexible notation and a mechanism to walk through all queues. In the last section we dealt with blocking actions. Because we already had a notation to describe multiple queues the extension to use it was fairly simple.

In the last chapter we dealt with blocked functions. When using multiple queues we had to provide a mechanism to deal with a blocked queue. Therefore we introduced a mechanism which is able to proceed with the next queue if a queue is fully blocked and when it is not the last queue.

We can conclude that our objective to extend cyclic strategic interleaving with multiple queues was a success. Using the proposed extension we created a base to continue with as ultimate goal: a full description to describe the Windows scheduler.

References


