Comments by Reviewer #1

- *1.) I don't see the point of working with a more general library of possibly infinite graphs when it is clear that no (necessarily finite) evolution of the state could ever generate an infinite graph.*

  It seemed the most convenient approach for us and we added the following comment on page 8 about justifying our choice. There was already a comment about having finiteness as a property, rather than a built-in assumption.

  > It seems for our purposes the most convenient representation of graphs are binary relations given by sets of pairs. The pairs stand for the edges in graphs. This relation-based representation has the advantage that the notions *waiting* and *holding* are already defined in terms of relations amongst threads and resources. Also, we can easily re-use the standard notions of transitive closure operations $\_*$ and $\_+$, as well as relation composition for our graphs.

- *2.) Later, the release function is defined using Hilbert choice (Isabelle/HOL's `SOME` function) as a method for emulating non-determinism. This is horrid. The highest level generation of traces handles non-determinism beautifully; using choice at a lower level lets you keep things functional, but hides the fact that you want to model non-determinism at the lower level as well. I agree that the final theorem does imply that the way in which the new waiting list is chosen is irrelevant, but this implication is only apparent through a close reading of that definition. Better I think to lift the non-determinism and make it more apparent at the top level.*

  > We generally agree with the reviewer about the use of `SOME`, but in this instance we cannot see a way to make this nondeterminism explicit without complicating the toplevel rules about *PIP* on page 12. By using `SOME`, we can leave implicit the order of the waiting queue returned by release (which corresponds to the original system call from Sha et al). If we represent the nondeterminism explicitly, we would need to add another argument to V specifying which thread is the next one that obtains the lock and add another premise to the *PIP* rule for ensuring that this thread is member of the waiting queue.

- *3.) In §4, there is an assumption made about the number of threads allowed to be created. Given the form of theorem 2, this seems to me to be unnecessary. It's certainly extremely weird and ugly because it says that there are at most BC many thread creation events in all possible future traces (of which there are of course infinitely many). ...*

We think this mis-reads our theorem: Although the constrains we put on thread creation prohibit the creation of higher priority threads, the creation of lower priority threads may still consume CPU time and prevent *th* from accomplishing its task. That is the purpose we put in the *BC* constraint to limit the overall number of thread creations. We slightly augmented the sentence on page 15 by:

> Otherwise our PIP scheduler could be "swamped" with *Create*-requests of lower priority threads.

- *Why are variables corresponding to resources given the name cs? This is confusing. rsc or r would be better.*

  cs stands for "critical section", which is the original name used by Sha et al to represent "critical resources".

- All other comments of the reviewer have been implemented.

Comments by Reviewer #2

- *Well-founded comment:* We updated the paragraph where *acyclic* and *well-founded* are used for the first time.

  Note that forests can have trees with infinite depth and containing nodes with infinitely many children. A *finite forest* is a forest whose underlying relation is well-founded and every node has finitely many children (is only finitely branching).

  We also added a footnote that we are using the standard definition of well-foundedness from Isabelle.

- *Comment about graph-library:* This point was also raised by Reviewer #1 and we gave a better justification on page 8 (see answer to first point of Reviewer #1).

- *20.) In the case of "Set th prio:", it is not declared what the cprec (e::s) th should be (presumably it is something like Max((prio,|s|),prec th s) or possibly just prec th (e::s) given the assumptions on Set events listed before).*

  We note the concern of the reviewer about the effect of the Set-operation on the *cprec* value of a thread. According to equation (6) on page 11, the *cprec* of a thread is determined by the precedence values in its subtree, while the Set operation only changes the precedence of the 'Set' thread. If the reviewer thinks we should add this explanation again, then we are happy to do so.

Comments by Reviewer #3

- *The verification is at the model level, instead of code level:...*

  The verification is indeed on the level of the algorithm. A verification of the C-code is *well* beyond the scope of the paper. For example, it would require a formalised semantics for C (as for example given in the seL4-project). This and interfacing with it would be a tremendous amount of work and we are not sure whether their results would actually sufficient for the code we wrote.

  In light of this, we have made it clearer in the abstract and in a footnote on the first page that our C-code is unverified. Additionally we added the following sentence in Section 5 before we describe the C-code:

  > While there is no formal connection between our formalisation and the C-code shown below, the results of the formalisation clearly shine through in the design of the code.

- *The model cannot express the execution of instructions. As a result, it is difficult to express the case that the critical region does infinite loops without generating events for system calls.*

  Yes, we discuss this limitation in the first paragraph of section 4 and already wrote that in this aspect we do not improve the limitations of the original paper by Sha et al.

- *The authors should have provided URL of their mechanized proofs for the review process.*

  As is custom, we have given a link to the sources. It is mentioned in the last sentence of the conclusion.

  The code of our formalisation can be downloaded from the Mercurial repository at `http://talisker.inf.kcl.ac.uk/cgi-bin/repos.cgi/pip`.

- *It is more like engineering work. It's unclear what general principles or theories are proposed.*

  The general point we are making is that a 'proof-by-hand' is generally worthless in this area for ensuring the correctness of an algorithm. We underline this point by listing the references [13, 14, 15, 20, 24, 25], which all repeat the error from the original paper.

  More specifically we extend the claims of Sha at al and give a machine-checked formalisation of our claims (the first such formalisation for PIP). We also wrote about our experiences:

> Following good experience in earlier work [27], our model of PIP is based on Paulson's inductive approach for protocol verification [18].

- *1) There are some places that the authors follow the Isabelle syntax, which require more explanation. For instance, in the definitions of "threads" and "priority" in page 5, I had trouble understanding the underscore until I realized that the 4 cases are pattern matching and they have orders.*

  We have added the sentence after the definition of *threads*:

  > We use _ to match any pattern, like in functional programming.

- *2) The last assumption on page 13: the assumption seems very strong. If you prove Theorem 1 inductively on all s (which means you have to consider s of length 1), then following the assumption you essentially require that the highest priority task is the first task created.*

  We feel like this mis-reads what we are proving and how we organised the statements. The thread *th* is the thread with the highest priority in the state *s*. It can be in *s* at any 'place'—it does not need to be the first one. What we prove is what happens to *th* in the state $s'@s$ which happens after *s*. We only require that threads created after *s* need to have a lower priority.

- *3) 2nd line of the proofs of Lemma 1: th should be th' ?*

  Yes, we corrected this error.

- *4) Is your state (event traces) finite or infinite?*

  Yes, they are always finite, but can be arbitrary long.

- *5) Assumption at the bottom of page 15: I don't understand why this assumption is necessary. First, is es a finite or infinite trace? If es is finite, of course there's limited number of Create-requests. Even if it can be infinite, I don't see how the PIP scheduler could be "swamped" with create requests. You already assumed that there are no threads of precedence higher than th created in es. If all the newly created threads have lower precedence, they have no chance to run anyway (because of the low precedence). Then why should we care?*

  Yes, *es* and *es*@*s* are finite traces (finite list of events), but they can be arbitrarily long.

  This means that knowing that es is finite, does *not* bound the number of create events.

We also understand that newly created threads have no chance to run (because of the lower priority), but the process of creating any processes consumes according to our model some time and therefore needs to be bounded. A Create event is not assumed to be ‘instantaneously’.

- *6) The next assumption in page 16 does not look right to me either. What if there are no actions of th’ in es at all, which may be caused by infinite loop inside the critical region of th’ (but the loop does not generate any events)? In this case, the assumption is still satisfied because the length is 0, which is less than BND(th’).*

  We made this point clearer (it was also requested by another reviewer). We discuss this limitation in more depth in the first paragraph of section 4 and already wrote that in this aspect we do not improve the limitations of the original paper by Sha et al.

- All other comments have been implemented.