Specifying concurrent components

Ian J. Hayes

July 19, 2017 15:57

Specification of talk

Me

pre	start on time
rely	you ask the right questions
guarantee	you'll understand my talk
assume	questions will eventually stop
post	finish talk

$\|^{1}$

You

pre	true
rely	I'll understand Ian's talk
guarantee	I'll ask the right questions
ensure	I'll stop asking questions eventually
post	finish listening

[1] Assuming Michael implements real-time scheduling for the parallel

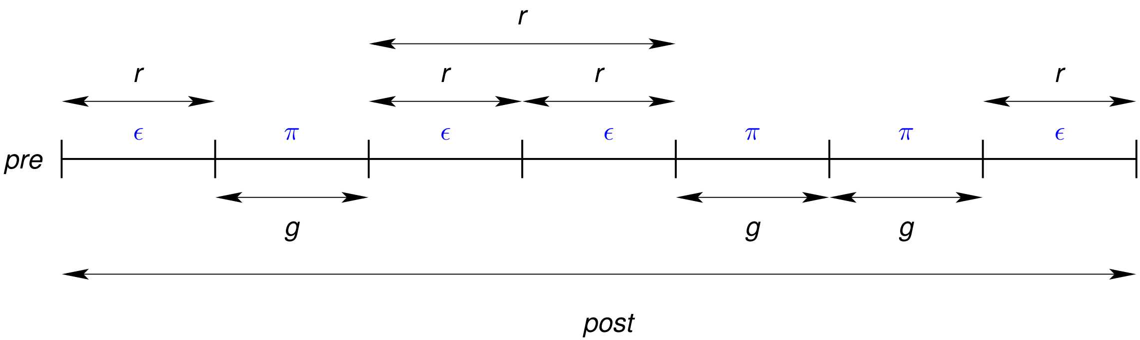
Overview

- Specification for the Linux ticket lock (and Peterson's) algorithm
- The specification emphasises decoupling
  - the specification of the module implementing the locking algorithm
  - from the context in which it is used,
  - allowing its development to take place independent of a particular context
- Need to clearly define the assumptions it makes about its context

Concurrency and interference (Jones [1])

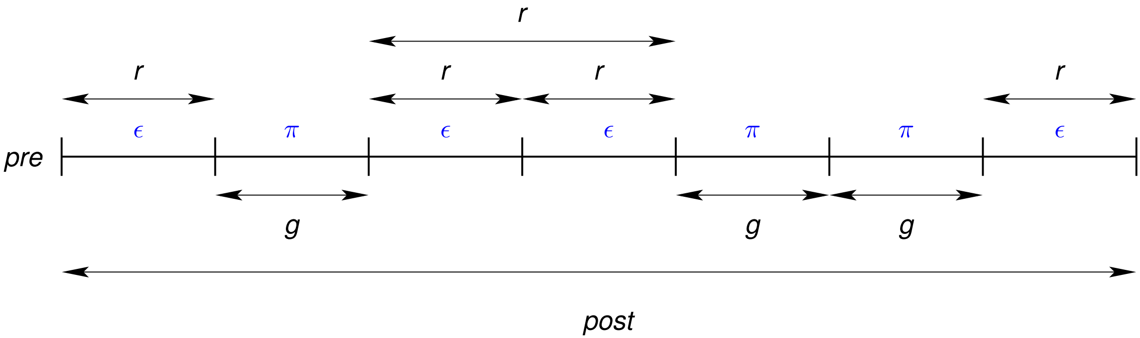
- pre condition
- post condition
- rely condition r
- guarantee condition g

- π atomic program step
- ϵ atomic environment step



Rely conditions

- If there is one environment step it must satisfy r
- If there are no environment steps, the interference is the identity relation, id, on states and hence $\mathrm{id} \Rrightarrow r$, i.e. r is reflexive
- If there are two environment steps, overall they satisfy $r \fatsemi r$ and hence $r \fatsemi r \Rrightarrow r$, i.e. r is transitive
- Hence $r^\star = r$



Progress of the environment

- To show termination of operations that may block (e.g. **await** b)
  - one needs to be able to assume that b will eventually be made (stably) true by the environment
- Because r is reflexive
  - it cannot be a well-founded relation and hence
  - it cannot be used to show that the environment eventually achieves some desired state
- The extension used here is to
  - add an additional assumption about the environment
  - expressed using temporal logic operators, like leads-to (e.g. $p_1 \leadsto p_2$).
  - these progress assumptions are in addition to the rely relation of Jones.

Linux ticket lock

- It is an interesting concurrent algorithm that can be used in a multiprocessor context where multiple processors are vying for access to a resource
- Processes requiring the resource first take a “ticket” (as in a bakery or deli) and are given access to the resource in ticket order
- Provided each process that gains access to the resource releases it, all processes with a ticket will eventually receive control of the resource

Lock Module Specification

- A module with two operations *acquire* and *release*
- The process at the head of the queue, if there is one, has the resource
- Processes are allocated the resource in queue order

module *Lock*$($*PID* : finite type with equality$)$
var $q : \mathrm{seq}\ PID$ **init** $q = [\,]$ **inv** $no\_duplicates(q)$

Acquiring the lock

The acquire operation gains access to the resource.

$acquire(k : PID) \mathrel{\widehat{=}}$

wr	q	// can only modify q
pre	$k \notin elems(q)$	// must not already be waiting in q
rely	$(hd(q) = k \Rightarrow hd(q') = k) \wedge$	// can't be preempted if k has resource
	$(k \in elems(q) \Rightarrow k \in elems(q'))$	// k stays in the queue
guar	$q' = q \vee$	
	$(k \notin elems(q) \wedge q' = q \frown [k])$	// may only append k (atomically)
assume	$k \in elems(q) \leadsto hd(q) = k$	// k gets to the head of queue
post	$hd(q') = k$	// k has the resource on termination

Releasing the lock

The *release* operation gives up control of the resource.

$release(k : PID) \mathrel{\widehat{=}}$

wr	q	// can only modify q
pre	$hd(q) = k$	// k must hold the resource
rely	$hd(q) = k \Rightarrow hd(q') = k$	// can't be preempted
guar	$q' = q \vee (hd(q) = k \wedge q' = tl(q))$	// can only remove k (atomically)
post	$k \notin elems(q)$	// it must release the resource

The set of process identifiers

- Any (large enough) set can be used for *PID*
- It is the responsibility of the calling context to ensure that
  - two different processes do not call *acquire* using the same process identifier
  - a process only calls *release* if it has the lock
- If the set of process identifiers is of size two, the specification can be implemented by Peterson's algorithm

$$PID\_Peterson ::= A \mid B$$
$$PetersonSpecification \mathrel{\widehat{=}} Lock(PID\_Peterson)$$

Example code for the use

$use\_lock(k : PID)\{$
$\{k \notin elems(q)\}$
$non\_critical\_section;$ // guarantees $R \wedge q' = q$
$\{k \notin elems(q)\}$
$acquire(k);$
$\{hd(q) = k\}$
$critical\_section;$ // relies on R, guarantees $q' = q$, and terminates
$\{hd(q) = k\}$
$release(k)$
$\{k \notin elems(q)\}$
$\}$

Use specification

$use\_lock(k : PID)$

wr $q, \ldots$

pre $k \notin elems(q) \wedge \ldots$

rely $(hd(q) = k \Rightarrow (hd(q') = k \wedge R)) \wedge$
$(k \in elems(q) \Rightarrow k \in elems(q'))$

assume $k \in elems(q) \rightsquigarrow hd(q) = k$

guar $((q = [\,] \vee hd(q) \neq k) \Rightarrow R) \wedge$
$(acquire.guarantee \vee release.guarantee)$

ensures $\forall j \in PID, i \in \mathbb{N} \bullet$
$j \in elems(q) - \{k\} \wedge hd(q) = k \wedge posn(q, j) = i \rightsquigarrow posn(q, j) < i$

post $k \notin elems(q) \wedge \ldots$

Top level

rely R

$$\{q = [\,]\} \quad \big\|_{k \in 1..N} \quad \ldots;\; use\_lock(k);\; \ldots$$

Correctness of progress property

One process is enabled

$$q = [\,] \lor (\exists\, k \in PID \bullet hd(q) = k)$$

For each process the ensures of all other processes imply its assumption

$$\forall\, k \in PID \bullet (\forall\, n \in PID \bullet n \neq k \Rightarrow ensure_n) \Rrightarrow assume_k$$

$$\begin{array}{l}
\forall\, k \in PID \bullet \\
\quad (\forall\, n \in PID \bullet n \neq k \Rightarrow \\
\quad\quad \forall\, j \in PID, i \in \mathbb{N} \bullet \\
\quad\quad\quad j \in elems(q) - \{n\} \land hd(q) = n \land posn(q,j) = i \leadsto posn(q,j) < i) \\
\quad \Rrightarrow \\
\quad \forall\, i \in \mathbb{N} \bullet k \in elems(q) \land hd(q) \neq k \land posn(q,k) = i \leadsto posn(q,k) < i
\end{array}$$

$$\begin{array}{l}
(\forall\, i \in \mathbb{N} \bullet k \in elems(q) \land hd(q) \neq k \land posn(q,k) = i \leadsto posn(q,k) < i) \\
\Rrightarrow \\
k \in elems(q) \leadsto hd(q) = k
\end{array}$$

Abstract implementation

module $\mathit{AbstractLock}(\mathit{PID} : \text{finite type with equality})$
var $q : \mathrm{seq}\ \mathit{PID}$ **init** $q = [\,]$ **inv** $\mathit{no\_duplicates}(q)$

Acquiring the lock

An abstract implementation of the *acquire* operation atomically adds k to the queue and then waits until k is at the head of the queue.

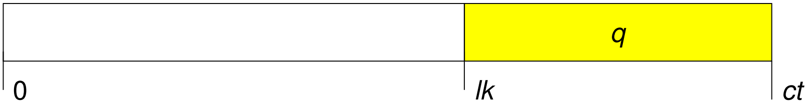
$$
\begin{array}{l}
\mathit{acquire}(k : \mathit{PID})\{ \\
\quad \{k \notin \mathit{elems}(q)\} \\
\quad \textbf{with}\ q\ \textbf{do}\ q := q \frown [k]; \\
\quad \textbf{await}\ k = \langle hd(q) \rangle \\
\}
\end{array}
$$

Releasing the lock

The *release* operation removes k from the head of the queue.

$$
\begin{array}{l}
\mathit{release}(k : \mathit{PID})\{ \\
\quad \{k = \mathit{hd}(q)\} \\
\quad \textbf{with } q \textbf{ do } q := \mathit{tl}(q) \\
\}
\end{array}
$$

Ticket lock implementation



The queue is empty if $lk = ct$. If the queue contains n elements then $lk + n = ct$.

module *TicketLock*$($*PID* : finite type with equality$)$
var $ct, lk : \mathbb{N}$ **init** $ct = 0 \wedge lk = 0$ **inv** $lk \leq ct$
abs-inv $\#q = ct - lk$

Acquiring the lock

The abstract *acquire* operation adds *k* to the end of the queue.

$$
\begin{array}{ll}
\mathit{acquire}(k : \mathit{PID})\{ & \\
\quad \textbf{var}\ \mathit{tk} : \mathbb{N}; & \\
\quad \textbf{with}\ \mathit{ct}\ \textbf{do}(\mathit{tk} := \mathit{ct};\ \mathit{ct} := \mathit{ct} + 1); & \text{// take a ticket} \\
 & \text{// } q(\mathit{tk} - \mathit{lk}) = k \\
\quad \textbf{await}\ \mathit{tk} = \langle \mathit{lk} \rangle & \text{// wait until } \mathit{lk} \text{ reaches the ticket's value} \\
\} &
\end{array}
$$

Releasing the lock

The abstract release operation removes the head of the queue.

$$
\begin{array}{l}
\mathit{release}(k : \mathit{PID})\{ \\
\ \langle \mathit{lk} \rangle := \mathit{lk} + 1 \\
\}
\end{array}
$$

Compare-and-swap

The **with** statement within *acquire* can be implemented using a compare-and-swap (CAS) instruction.

$$CAS(x, old, new, done) \mathrel{\widehat{=}} \textbf{with}\ x\ \textbf{do}(\textbf{if}\ x = old\quad \begin{array}{l}\textbf{then}\ x := new;\ done := true\\ \textbf{else}\ done := false)\end{array}$$

Taking ticket using CAS

Because an execution of a compare-and-swap can fail, it is placed in a loop that repeats until it succeeds.

var *done* : $\mathbb{B}$;
repeat
  $tk := \langle ct \rangle$;
  $CAS(ct, tk, tk + 1, done)$
until *done*

Unfortunately this loop may never terminate.

Peterson's algorithm

module *PetersonLock* **implements** *PetersonSpecification*

var $req : PID \to \mathbb{B}$ **init** $req(A) = req(B) = false$

$tk : PID$ **init** $tk \in PID$

Acquiring the lock

The complement of a process identifier, $\overline{k}$, gives the identifier of the other process, i.e. $\overline{A} = B$ and $\overline{B} = A$.

$$
\begin{array}{l}
acquire(k : PID)\{ \\
\quad req(k) := true;\ tk := \overline{k}; \\
\quad \textbf{await}\ req(\overline{k}) \Rightarrow tk = k \\
\}
\end{array}
$$

Releasing the lock

$$\mathit{release}(k : \mathit{PID})\{$$
$$\quad \mathit{req}(k) := \mathit{false}$$
$$\}$$

For an application using calls to $\mathit{acquire}/\mathit{release}$ with a constant process identifiers, e.g. $\mathit{acquire}(A)$, one can replace *req* by two variables *a* and *b*, corresponding to $\mathit{req}(A)$ and $\mathit{req}(B)$, respectively. That is, one can just inline the call and optimise it for the particular, constant parameter.

Conclusions

- compositionality
- modularisation
- abstract specification of operations
- data refinement

References

C.B. Jones.
Tentative steps toward a development method for interfering programs.
ACM ToPLaS, 5(4):596–619, October 1983.