A partial barrier based on fetch-and-increment

Wim H. Hesselink, whh599

September 25, 2023

Abstract

A partial barrier for size M is a synchronization mechanism for thread systems that allows threads repeatedly to pass an *entry* point only in batches of precisely M elements. The paper presents the design of a partial barrier, based on the hardware primitive fetch-and-increment. The correctness of the design is proved by means of invariants and a refinement function towards a partial barrier prototype.

keywords: synchronization, barrier, fetch-and-increment

1 Introduction

In a system of N threads, a barrier is a synchronization mechanism to let the threads repeatedly pass through the call of a barrier. When a thread arrives at the barrier, it has to wait for all threads to arrive at the barrier. The threads can pass the barrier only when all have arrived.

A *partial barrier* for a number M is a variation of this idea. It only allows the threads to pass in batches of M threads. All threads compete in a loop and repeatedly arrive at the "barrier", which selects a new batch of M threads from those that have arrived. When this batch has been launched and subsequently all its members have been released, a new batch of M threads can be launched, as soon as available.

In general, a partial barrier is not a barrier. It can serve as a barrier only if M is the total number of threads. At the other extreme, the case of $M = 1$ is mutual exclusion, compare [2] and the references given there. Indeed, the partial barrier looks like a special case of group mutual exclusion, see [1] and its references.

To investigate its behavior, the partial barrier is put into its lifeline loop, just as with mutual exclusion. Indeed, every thread is in an unbounded loop

(0)
```
loop of thread p :
    NCS ;
    entry(p) ;
    CS ;
    release(p)
end loop .
```

Here *NCS* stands for the noncritical section, *CS* for the critical section, both are program fragments that do not concern us. The partial barrier is the pair of functions *entry*, *release*. In the function *entry*, the threads have to wait until a batch of M members has been formed and can be launched.

The aim of this paper is to construct a partial barrier and to verify its correctness. For the correctness proof, a formal specification has to be developed first. Therefore Section 2 presents two partial barriers: an abstract prototype and an implementation with fetch-and-increment instructions and atomic registers. Section 3 contains the verification that the implementation refines the prototype and therefore satisfies the specification. Section 4 draws conclusions.

2 Two partial barriers

In Section 2.1, a simple state machine is constructed as a general prototype. Any system that refines this machine is a partial barrier. Section 2.2 presents an implementation of a partial barrier constructed with fetch-and-increment instructions.

2.1 A partial barrier formally specified

What are the requirements for partial barriers? It is clearly required that the number of threads in *CS* must be $\leq M$. This, however, is not enough, because it would allow new threads to enter *CS* when part of the previous batch have not yet been released.

The easiest way to specify a partial barrier is to require that its lifetime loop implements (i.e., simulates) the following prototype, where W is the set of waiting threads and B is the batch.

shared variables :
  W, B : **set of thread** ;
initially: $W = B = \emptyset$.

loop of thread p :
1 *NCS* ;
2 **add** p **to** W ;
3 **await** $p \in B$;
4 *CS* ;
5 **remove** p **from** B
end loop .

Here lines 2 and 3 represent *entry*, and line 5 represents *release*. Repeatedly, the sets W and B are modified by an auxiliary thread, the *butler*:

loop of thread *butler* :
  $\langle$ **if** $B = \emptyset \ \wedge \ \#W \geq M$ **then**
      choose $B \subseteq W$ **with** $\#B = M$;
      $W := W \setminus B$;
    endif $\rangle$
end loop .

Here the brackets $\langle\ \rangle$ enclose a single atomic transition. For a finite set S, the expression $\#S$ denotes its cardinality, its number of elements. So, the *butler* can act only if B is empty and at least M threads are waiting, it then moves M threads from W to B.

2.2 An algorithm with fetch-and-increment

The following implementation of the partial barrier is proposed. The algorithm uses three shared integer variables, and one integer local variable. For now, assume that the integers are unbounded. The hardware instruction *fetch-and-increment*($\mathtt{x}, n$) reads the value of the shared variable `x` and assigns $\mathtt{x} := \mathtt{x} + n$ without interference.

int $\mathtt{free} = 0, \mathtt{high} = 0, \mathtt{inCS} = 0$.

entry(*thread p*) :
  int ticket := *fetch-and-increment*(`free`, +1) ;
  await(*ticket* $\leq$ `high`) ;
  if *ticket* = `high` **then**
    await(`inCS` = 0) ;
    await(*ticket* + $M \leq$ `free`) ;
    `inCS` := M ;

```
        high := ticket + M
    endif
end

release(thread p) :
    fetch-and-increment(inCS, −1)
end .
```

The algorithm needs three await statements, one to controle the size of the batch that is being selected, one to decide that the previous batch has been released completely, and one to decide that enough threads are available.

In Section 3.4, it is shown that the algorithm can be used with integers bounded by a constant MAX, provided the following requirements are met. The expression $MAX + 1$ gives the smallest integer MIN. The number of active threads is less than MAX, and $MIN < -M$. The two inequalities $x \leq y$ in the await statements have to be implemented (i.e., replaced) by $0 \leq y - x$.

2.3 The lifeline loop of the implementation

To prove the correctness of the algorithm of Section 2.2, it is put into the lifeline loop (0). For the sake of the verification, an integer ghost variable `low` is introduced.

```
loop of thread p :
11    NCS ;
12    ticket_p := free ; free := free + 1 ;
13    await (ticket_p ≤ high) ;
14    if ticket_p = high then
15        await (inCS = 0) ;
16        await (ticket_p + M ≤ free) ;
17        inCS := M ;
18        high := ticket_p + M ;
      endif ;
19    skip ;
20    CS ;
21    inCS := inCS − 1 ; low := low + 1
end loop .
```

Line 19 contains a superfluous operation *skip*, a stuttering step, to help in the construction of a refinement function. The initial condition of the lifeline loop is

$$\texttt{free} = \texttt{high} = \texttt{inCS} = \texttt{low} = 0 \ \wedge \ \forall\, q : pc_q = 11 \ .$$

3 The formal verification

Section 3.1 recalls the concepts of state machines, executions and invariants, as well as a method to collect and prove invariants for a given state machine. Section 3.2 describes the prototype of Section 2.1 as a state machine. In Section 3.3, the state machine for the algorithm of Section 2.3 is described. The method of Section 3.1 is used to collect and prove several invariants of the algorithm. Finally, Section 3.5 presents and proves a refinement from the algorithm to the prototype.

3.1 State machines

As we are interested only in safety properties, the state machines are kept simple, i.e., without liveness properties. A state machine is therefore defined as a tuple $K = (X, Y, N, obs)$ where X is a set, the state space, $Y \subseteq X$ is the initial condition, N is a reflexive relation on the state space, the next-state relation, and obs is a function on X.

An execution of machine K is an infinite sequence $x_0, \ldots$ of states such that $x_0 \in Y$ and $(x_i, x_{i+1}) \in N$ for every natural number i. It is observed via the observation sequence $(obs(x_i))_i$. By a predicate P, we mean a boolean function on X, or equivalently the subset of X where P holds. Predicate P is said to be *invariant* iff it contains every state of every execution.

A subrelation $S \subseteq N$ is called a *command*. If S is a command and P and Q are predicates on X, the so-called Hoare triple $\{P\}\ S\ \{Q\}$ expresses that command S transforms every state in P into a state in Q, formally $x \in P \wedge (x, y) \in S \ \Rightarrow\ y \in Q$.

We need a bit of theory about invariants from [3]. A predicate P is said to be *preserved* by command S iff $\{P\}\ S\ \{P\}$ holds. Predicate P is called *stable* if it is preserved by relation N. A predicate is called *inductive* iff it is stable and holds initially. Every inductive predicate is an invariant. A predicate implied by an inductive predicate is an invariant.

Predicate P is said to be *threatened* by a command S iff it is not preserved by S. If predicate P is threatened by command S, a predicate Q is called a *remedy* for P and S iff $\{P \wedge Q\}\ S\ \{P\}$. Let C be a set of commands such that $N = \bigcup C$. If one has a family of predicates such that any member of the family that is threatened by a command in C, has some remedy that is a conjunction of members of the family, then the conjunction of the family is stable. If moreover all members hold initially, the conjunction is inductive, and each member of the family is an invariant. This is the method used below to obtain and prove invariants.

3.2 The prototype as a state machine

Let *Thread* be the type of the threads, i.e., the thread identifiers. It can be a subset of the natural numbers. The protoype of Section 2.1 is formalized as a machine $K = (X, Y, N, obs)$ as follows. The state space X is spanned by the variables W and B, both holding sets of threads, and the program counters $pc : Thread \to \mathbb{N}$. The initial condition Y is given by

$$W = B = \emptyset \ \wedge\ \forall\, q \in Thread : pc_q = 1 \ .$$

For every thread q, and every line number $\ell \in \{1, 2, 3, 4, 5\}$, let $N_{\ell,q}$ be the relation that contains the pairs $(x, y) \in X^2$ such that state y is obtained from state x by executing the command of line ℓ of the loop of thread q. For the cases $\ell \in \{1, 3, 4\}$, the command only increments the value of pc_q. Line 3 is executed only if $q \in B$. Let N_ℓ be the union of the sets $N_{\ell,q}$ for all threads q. Let N_0 be the set of the pairs (x, y) such that state y is obtained from state x by execution of the *Butler* thread. The next state relation N is the union $N = \mathbf{1} \cup N_0 \cup \bigcup_\ell N_\ell$, where $\mathbf{1}$ is the identity relation, the set of all pairs (x, x) with $x \in X$.

Finally, it is observable which threads are in *NCS* or *CS*. As 1 and 4 are the locations of *NCS* and *CS*, respectively, the observation function $obs : X \to (Thread \to \mathbb{N})$ is given by

$$obs(x)(q) = (x.pc_q = 1\ ?\ 1 : x.pc_q = 4\ ?\ 2 : 0) \ .$$

Here, pc is treated as a field of state x.

3.3 The implementation as a state machine

The algorithm of Section 2.3 is formalized in the state machine $K' = (X', Y', N', obs')$. The state space X' is spanned by the shared variables `free`, `high`, `inCS`, the shared ghost variable `low`, and the private and local variables pc_q and $ticket_q$ for all threads q. Just as with the prototype of Section 3.2, the step relation N' is built in terms of the steps of the threads at the lines ℓ of (2.3). The step of thread q at line ℓ is represented by the relation $N'_{\ell,q}$. The steps at line ℓ for all threads q together are represented by the union $N'_\ell = \bigcup_q N'_{\ell,q}$. The step relation is itself $N' = \mathbf{1} \cup \bigcup_\ell N'_\ell$ where ℓ ranges from 11 to 21. As *NCS* and *CS* in (2.3) are at the lines 11 and 20, respectively, the observation function obs' on X' is given by

$$obs'(x)(q) = (x.pc_q = 11\ ?\ 1 : x.pc_q = 20\ ?\ 2 : 0) \ .$$

We need several invariants for machine K'. The method described in Section 3.1 is used to collect and prove invariants. The proof assistant PVS of [4] was used to determine threatened predicates and

potential remedies. For numbers i and k, one defines the sets of threads $[i] = \{q \mid pc_q = i\}$ and $[i,k] = \{q \mid i \le pc_q \le k\}$.

A thread is said to be *active* iff it is in $[13,21]$. The number of active threads is given by the invariant

Iq1 : $\#[13,21] = \texttt{free} - \texttt{low}$.

Indeed, this predicate is inductive: initially, both sides are zero; when some thread executes line 12, it enters the set $[13,21]$ and increments `free`; when it executes line 21, it leaves $[13,21]$ and increments `low`.

For the tickets we have the invariants

Iq2 : $q, r \in [13,21] \ \wedge \ ticket_q = ticket_r \ \Rightarrow \ q = r$,
Iq3 : $q \in [13,21] \ \Rightarrow \ ticket_q < \texttt{free}$.

Predicate *Iq2* is threatened only by step 12. It has the remedy *Iq3*. Predicate *Iq3* is inductive.

All tickets above `high` are still active:

Iq4 : $\texttt{high} \le n < \texttt{free} \ \Rightarrow \ \exists\, q \in [13,21] : ticket_q = n$.

This predicate is threatened only by the steps 18 and 21. It has the respective remedies

Iq5 : $q \in [15,18] \ \Rightarrow \ ticket_q = \texttt{high}$,
Iq6 : $q \in [19,21] \ \Rightarrow \ ticket_q < \texttt{high}$.

Predicate *Iq5* is threatened only by step 18. It has the remedy *Iq2*.

Predicate *Iq6* is threatened only by the steps 14 and 18. At step 18, it has the remedy *Iq5*. At step 14, it has the remedy

Iq7 : $q \in [14] \ \Rightarrow \ ticket_q \le \texttt{high}$.

Predicate *Iq7* is threatened only by step 18. It has the remedy *Iq5*.

The set $[13,21]$ is the disjoint union of the two sets

(1) $Batch = \{q \mid q \in [13,21] \ \wedge \ ticket_q < \texttt{high}\}$,
$Waiting = \{q \mid q \in [13,21] \ \wedge \ \texttt{high} \le ticket_q\}$.

The number of elements of *Waiting* is determined by

Crit1 : $\#Waiting = \texttt{free} - \texttt{high}$.

This follows easily from *Iq2*, *Iq3*, *Iq4*, together with the new invariant

Jq1 : $\texttt{high} \le \texttt{free}$.

Predicate *Jq1* is threatened only by step 18. It has the remedy

Jq2 : $q \in [17,18] \ \Rightarrow \ ticket_q + M \le \texttt{free}$.

Predicate *Jq2* is inductive.

Using *Iq1* and *Crit1*, one easily obtains

Crit2 : $\#Batch = \texttt{high} - \texttt{low}$.

This implies that the set *Batch* is empty if and only if $\texttt{high} = \texttt{low}$. The difference between `high` and `low` is governed by the two invariants:

Jq3 : $(\texttt{high} = \texttt{low} + \texttt{inCS}) \ \vee \ (\exists\, q : q \in [18])$,
Jq4 : $q \in [16,18] \ \Rightarrow \ \texttt{high} = \texttt{low}$.

The conjunction of *Jq3*, *Jq4* and *Crit2* implies

Jq3A : $(\#Batch = inCS) \ \vee \ (\exists\, q : q \in [18])$,
Jq4A : $q \in [16,18] \ \Rightarrow \ Batch = \emptyset$.

So the size of *Batch* is reflected in the value of `inCS`.

Predicate *Jq3* is threatened only by step 18. It has the remedies *Iq5* and the new predicate

$$Jq5: \quad q \in [18] \;\Rightarrow\; \texttt{inCS} = M \;.$$

Predicate *Jq4* is threatened only by the steps 16 and 18. It has the remedies *Jq3* and the new predicate

$$Iq2A: \quad q, r \in [15, 18] \;\Rightarrow\; q = r \;.$$

Predicate *Iq2A* is implied by *Iq2* and *Iq5*.

Predicate *Jq5* is threatened only by step 21. It has the remedies *Jq4A* and

$$Jq6: \quad q \in [19, 21] \;\Rightarrow\; q \in Batch \;.$$

Predicate *Jq6* is threatened only by the steps 14 and 18. It has the remedies *Jq4A*, *Iq7*. This concludes the construction and proof of the invariants for the partial barrier.

Absence of deadlock

Absence of deadlock can be proved in the following form:

Theorem 1 *Assume that the number of active threads is at least M. Then there is an active thread that can do a step.*

Proof. By *Iq1*, the assumption implies that $\texttt{low} + M \leq \texttt{free}$. If $q \in Batch$, then it is active and *Iq5* implies that it is in not in $[15, 18]$. Therefore, it can do a step. One can therefore assume that *Batch* is empty. Then *Crit2* implies that $\texttt{high} = \texttt{low}$. This implies that $\texttt{high} + M \leq \texttt{free}$. As $M > 0$, predicate *Iq4* implies that there is an active thread p with $ticket_p = \texttt{high}$. If thread p is at line 13 or 16, it is enabled. If it is a line 15, it is also enabled because of *Jq3* and *Iq2A*. □

Every step of an active thread means progress, because the algorithm has no other loops than the main loop that contains *NCS* and *CS*.

3.4 Overflow

The above verification of the algorithm assumes unbounded integers, but the integers on an actual computer are bounded, and will give overflow when the incrementations in line 12 accumulate. This may affect the comparisons in the lines 13, 14, and 15. For the taming of the overflow, a new invariant is postulated:

$$Jq7: \quad q \in [13, 21] \;\Rightarrow\; \texttt{high} - M \leq ticket_q \;.$$

This predicate is threatened only by step 18. It has the remedies *Jq1*, and *Iq5* and *Jq4A*.

Assume that the total number of threads is bounded by N. Then *Crit1* implies that $\texttt{free} - \texttt{high} \leq N$. Using *Iq3* and *Jq7*, it follows that the active threads satisfy

$$-M \leq ticket_q - \texttt{high} < N \;,$$
$$-M < \texttt{free} - ticket_q - M \leq N \;.$$

If the constants $-M$ and N give no overflow, the exprssions $ticket_q - \texttt{high}$ and $\texttt{free} - ticket_q - M$ also give no overflow. These expressions are sufficient for the guards in the lines 13, 14, 15. It therefore suffices to replace the inequalities $x \leq y$ and $x = y$ by $0 \leq y - x$ and $0 = y - x$, respectively.

3.5 Refinement

The machines of the prototype and the implementation are now related by means of a refinement function. In general, for any pair of machines $K = (X, Y, N, obs)$ and $K' = (X', Y', N', obs')$, a function $f : X' \to X$ is called a *refinement function* from K' to K iff the machine K' has an invariant J such that

$$\begin{array}{ll} C1: & \forall\, x \in X' : x \in Y' \;\Rightarrow\; f(x) \in Y \;, \\ C2: & \forall\, x_1, x_2 \in X' : x_1 \in J \;\wedge\; (x_1, x_2) \in N' \;\Rightarrow\; (f(x_1), f(x_2)) \in N \;, \\ C3: & \forall\, x \in X' : x \in J \;\Rightarrow\; obs(f(x)) = obs'(x) \;. \end{array}$$

The relevance of this definition is that every execution of K' is mapped to an execution of K, and that every observation sequence of K' is an observation sequence of K.

For the state machines K and K' of the Sections 3.2 and 3.3, respectively, a refinement function f from K' to K is constructed as follows. First, the sets *Batch* and *Waiting* of Formula (1) depend of the state $x \in X'$; this dependance is made explicit by writing $Batch(x)$ and $Waiting(x)$, respectively. Now the function f maps the state $x \in X'$ to the state $y = f(x)$ in X with the fields $y.W = Waiting(x)$ and $y.B = Batch(x)$ and $y.pc_q = g(x.pc_q)$ where

$$\begin{array}{rl} g(\,\ell) = (\ell = 12\,? & \;\; 2 \\ \quad : 13 \le \ell \le 19\,? & \;\; 3 \\ \quad :\; \ell = 20\,? & \;\; 4 \\ \quad :\; \ell = 21\,? & \;\; 5 : \;\; 1) \;. \end{array}$$

The verifications of the conditions *C1* and *C3* are trivial.

It remains to verify *C2*. Let J be the conjunction of all invariants obtained in Section 3.3. Let $(x_1, x_2) \in N'$. If it is an identity step $x_1 = x_2$ of machine K', then $f(x_1) = f(x_2)$ is an identity step of machine K. Therefore, assume that it is a step of thread p at line ℓ. Step 11 of K' does not affect the sets *Waiting* and *Batch*, and corresponds to step 1 of K. Step 12 of K' adds thread p to *Waiting* because of the invariant *Jq1*; this corresponds to step 2 of thread p in machine K. The steps ℓ with $13 \le \ell \le 17$ do not modify `high` or *ticket*, and therefore keep *Waiting* and *Batch* constant; they correspond to identity steps of K. If thread p does step 18, it satisfies the precondition $Batch = \emptyset$ because of *Jq4A*. It transfers M elements from *Waiting* to *Batch* because of *Iq2*, *Iq4*, and *Jq2*. Therefore, it corresponds to the *butler* step in K. If thread p does step 19, it has the precondition $ticket_p <$ `high` because of *Iq6*. Therefore this step corresponds to step 3 of thread p in K. Finally, it is easy to see that the steps 20 and 21 of K' correspond to the steps 4 and 5 of K, respectively.

Remark. The *skip* step of line 19 of K' corresponds to the **await** step of line 3 in K. If one does not introduce the *skip* step, one is forced to map step 18 of K' to a combination of the *butler* step with the **await** step of K.

4 Conclusion

The easiest specification of the partial barrier is by means of a prototype state machine. A system is or implements a partial barrier if and only if it refines the prototype. A partial barrier can be implemented by atomic read/write integers with a fetch-and-increment instruction.

References

[1] A. A. Aravind and W. H. Hesselink. Group mutual exclusion by fetch-and-increment. *ACM Transaction on Parallel Computing*, 5(4), 2019. Article number 14.

[2] P. A. Buhr, D. Dice, and W. H. Hesselink. High-performance N-thread software solutions for mutual exclusion. *Concurrency Computat.: Pract. Exper.*, 27:651–701, 2015. http://dx.doi.org/10.1002/cpe.3263.

[3] W. H. Hesselink. Trylock, a case for temporal logic and eternity variables. *Science of Computer Programming*, 216(102767), 2022.

[4] S. Owre, N. Shankar, J.M. Rushby, and D.W.J. Stringer-Calvert. *PVS Version 7.1, System Guide, Prover Guide, PVS Language Reference*, 2020. `http://pvs.csl.sri.com`, accessed 1 Dec. 2021.