For a couple of years after my discovery of the Bakery Algorithm, everything I learned about concurrency came from studying it.

Leslie Lamport 1

We give here a step-by-step development of Lamport’s Bakery Algorithm 2 for many-process mutual exclusion, and its proof, using three processes as an example. The three-process case is a model for cases with more processes.

The two-process case is substantially simpler, which is why we skip over it: and in any case the queue-based metaphor works for that already. (Why can’t we extend Misra’s queue-based approach to three processes?)

• General setting and the real-world analogy

Like the amoeba-based approach to Dijkstra’s shortest-path algorithm (Reynolds) and our queue-based approach to Peterson’s two-process mutual-exclusion algorithm (Misra), 3 we approach the more general mutual-exclusion problem via a real-world analogy. A bakery shop has only one server, who serves only one client at a time; but there are many potential clients in the shop. Mutual exclusion of the server is organised by a ticket machine that gives each client in turn the next-higher number; and the server serves clients in order of ticket number.

That sounds reasonable (and indeed it works, as we will see). But you should immediately be suspicious: Client C’s accessing the ticket machine must surely be something like $n_C, n := n, n+1$ to give the current ticket number to C and to increment the machine’s number. And doesn’t that require mutually exclusive access to the ticket machine? It’s instructive to see how that apparent circularity is avoided; so keep your eyes on it.

• First coding attempt

Call our three processes $P, Q, R$ and let their ticket variables be $p, q, r$, natural numbers representing queue positions, with value 0 however meaning “not in the queue at all.” We now work our way through a number of coding attempts; but we do not reach a solution until Fig. 7.

Before we begin, we note that it has been shown to be impossible to give a fully symmetric solution to the mutual exclusion problem: 4 there must be some built-in tie-breaking convention. In our case we will code it explicitly by using strict less-than between $p, q, r$ just when they are out-of-order alphabetically: thus in effect we will favour Process $P$ over Process $Q$ when their two “tickets” $p, q$ are equal.

---

1 //research.microsoft.com/en-us/um/people/lamport/pubs/pubs.html
Leslie Lamport is the computer scientist who built LATEX on top of Donald Knuth’s TEx.
Until then he was much better known for his influential and foundational contributions to concurrency and how to reason about it.


3 It’s possible we did neither of those in 2019.

Non-negative integers \( p, q, r \) are tickets for \( P, Q, R \) respectively.

\[
\begin{array}{ccc}
\text{integer } p & \text{integer } q & \text{integer } r \\
\text{Initially: } p = q = r = 0
\end{array}
\]

Simulate taking next-higher ticket: symbol \( \sqcup \) means maximum.

Variables \( q_P, r_P \) are Process \( P \)'s copies of Process \( Q, R \)'s tickets.

\[
\begin{align*}
q_P & := q \\
r_P & := r \\
p := 1 + q_P \sqcup r_P & = 1 + r_P \sqcup q_P
\end{align*}
\]

Wait until holding the lowest ticket value.

\[
\begin{align*}
\text{await } p \leq q \vee q=0 & & \text{await } q \leq r \vee r=0 & \text{await } r < p \vee p=0 \\
\text{await } p \leq r \vee r=0 & & \text{await } q < p \vee p=0 & \text{await } r < q \vee q=0 \\
\text{CS} & & \text{CS} & \text{CS}
\end{align*}
\]

Leave the bakery.

\[
\begin{align*}
p & := 0 \\
q & := 0 \\
r & := 0
\end{align*}
\]

The code segments given are implicitly within unending loops, one for each process. The code await \( p \leq q \vee q=0 \) for example delays \( P \) until either it is ahead of \( Q \) in the queue or \( Q \) is not in the queue at all.

Figure 1: First attempt at three-way Bakery Algorithm.

A correct version is given in Fig. 7.

\textit{Other business,} that is not contending for the critical section, is abbreviated \( OB \); and \textit{critical section} is abbreviated \( CS \). They are similar in that neither may access variables used for the mutual exclusion; they differ in that a process is not allowed to stay in \( CS \) forever, whereas its stay in \( OB \) is not limited.

The variables \( p, q, r \) are globally readable, but are written-to only by their “owning” processes \( P, Q, R \) respectively. Other variables are fully local in the sense that they are readable and writable only by the process indicated with a subscript. Thus \( p \) is written-to only by \( P \) although it can be read by \( Q, R \); but \( q_P, r_P \) are written \textit{and} read only by \( P \).\(^5\)

Fig. 1 is a first attempt at the algorithm, inspired directly by the bakery analogy. Remember that ticket value 0 means “not in the queue,” whereas a positive value gives the (sparse-) queue position, with ties broken as above.

\(^5\) This convention is broken by global variables \( b_{P, Q, R} \) added later in Fig. 5: at that stage, however, we should have absorbed what’s going on.
We add an assertion $A$ in Process $P$: if it is globally correct, it will still be true at $P$’s critical section. The only significant interference comes from $q:=1+rQ\sqcup pQ$ in $Q$, the statement that is motivated by the action “set $q$ to a value strictly higher than $p$.” But it does not access $p$ directly to do this (why not?) Instead it accesses its recent copy $q_P$ of $p$, and so we need to be sure that $q$’s “being more than $q_P$” implies its “being more than $p$.”

So we add an assertion $B$ in Process $Q$ just before its first assignment to $q$, to prevent its interference with Assertion $A$; the new assertion is locally correct, being established by the earlier $pQ:=p$.

The assertion $p\neq 0$ at $P$’s critical section is trivially correct (both locally and globally), established earlier and not interfered with by either of $Q, R$.

Now with similar assertions in $Q$ we would have

$$p\neq 0 \land (p\leq q \lor q=0) \land q\neq 0 \land (q<p \lor p=0)$$

if $P, Q$ were at $CS$ simultaneously, the contradiction we are looking for.

Still we have potential interference of $p:=1+q_P\sqcup r_P$ in $P$ which, by increasing $p$, can falsify Assertion $B$ in $Q$, and indeed the algorithm is unsafe: execute $P$ up to and including $\rightarrow$, then $Q$ up to $\Rightarrow$, then $P$ again up to $\Rightarrow$.

Figure 2: First proof attempt for three-way Bakery Algorithm.
• First proof attempt

In Fig. 2 we concentrate on the interference Process Q might cause to Process P, since the other interactions will be analogous. We add Assertion A in Process P with the aim of its (still) holding in the critical section CS, two statements later; we hope it will be mutually inconsistent with the similar assertions we could establish for Q, R, since that would establish safety.

To show global correctness of Assertion A in Process P, we need to add Assertion B before the potentially interfering assignment to q in to Process Q; it is locally correct in Q from the earlier assignment to pQ.

To show global correctness of B however, we must look back again at Process P, at its assignments to p. The second assignment p := 0 is no problem, since we know 0 ≤ pQ. But since Process P does not mention pQ in its first assignment to p, yet can affect p in essentially an arbitrary way (depending e.g. on pR), there seems no way to prevent its interference with Assertion B, while using only the variables we have.

And indeed the algorithm of Fig. 2 is incorrect. Allow Process P to execute up to (and including) →; then execute Q up to ⇒; finally, continue P up to ⇒.

\[
p = q = r = 0
\]

\[
\begin{align*}
OB & \quad OB & \quad OB \\
qu_p := q & \quad r_Q := r & \quad p_R := p \\
r_R := r & \quad pQ := p & \\
q := 1 + q_p \land q & \quad q := 1 + r_Q \land p_Q & \quad r := 1 + p_R \lor q_R \\
{\{B: p \leq pQ \lor \alpha\}} & \quad {\{A: \neg \alpha \land (p \leq q \lor q = 0)\}} & \\
\text{await } p \leq q \lor q = 0 & \quad \text{await } q \leq r \lor r = 0 & \quad \text{await } r < q \lor q = 0 \\
{\{p \neq 0\}} & \quad CS & \quad CS \\
\text{await } p \neq r \lor r = 0 & \quad q = 0 & \quad r := 0 \\
\{p \neq 0\} & \\
CS & \\
P := 0 & \\
\end{align*}
\]

We add label \( \alpha \) in an attempt to exploit the relative positions of Assertions A, B in the code of Processes P, Q. Both A and B are altered to make use of the new label.

Figure 3: Second proof attempt for three-way Bakery Algorithm.

• Second proof attempt

In Fig. 3 we add label \( \alpha \) to try to get hold of the fact that \( p \) cannot be (or, rather, should not be) assigned-to in \( P \) while we’re trying to establish global

\footnote{Accessing \( p \) directly would be feasible for two-process mutual exclusion; but for three we would need \( q := 1 + r \land p \), and that cannot be implemented with a single remote access. See Footnote 14.}
\[ p = q = r = 0 \]

<table>
<thead>
<tr>
<th>OB</th>
<th>( q := q )</th>
<th>( r := r )</th>
<th>( p := p )</th>
</tr>
</thead>
</table>

\[ p := 1 + q \uplus r \quad \{ p \neq 0 \} \]

\[ \beta : \{ B : \beta \land (p \leq p q \lor \alpha) \} \]

| \( \alpha : \text{await } \neg \beta \) | \( q := 1 + q \uplus p q \) | \( r := 1 + p \uplus q R \) |

\[ \text{await } p \leq q \lor q = 0 \]
\[ \{ A : \neg \alpha \land (p \leq q \lor q = 0) \} \]

\[ \text{await } q \leq r \lor r = 0 \]

\[ \text{await } r < p \lor p = 0 \]

\[ \text{await } r < q \lor q = 0 \]

\[ \{ p \neq 0 \} \text{ CS} \]

\[ \text{CS} \]

\[ \text{CS} \]

\[ p := 0 \quad q := 0 \quad r := 0 \]

We add an \text{await } \neg \beta to prevent falsification of \( \alpha \) while attempting to achieve global correctness of \( B \).

Figure 4: Third proof attempt for three-way Bakery Algorithm.

Correctness of \( B \). Note however that the algorithm in Fig. 2 is incorrect—so we’re not hoping to fix the proof entirely. What we’re hoping for is that our attempts to fix the proof will tell us how we must alter the algorithm.

This last point is extremely important.

Ad hoc methods of “fixing” problems in concurrent algorithms are usually based on operational reasoning that introduces new features to prevent a certain path from being followed. They do not work; instead, they merely make the problem more obscure, moving it to another place where it might be found by your successor—if you’re lucky. The only sure method, though it takes more skill, is to try to fix the proof and look for new features in the algorithm that will make the proof go through.

Fig. 3 indeed looks more promising. Global correctness of \( A \) is guaranteed by \( B \) as before; but global correctness of \( B \) itself is now assisted by the added disjunct \( \alpha \). That is, although the assignment to \( p \) in \( P \) could have invalidated the old \( B \), it cannot invalidate the new one since executing that assignment makes \( \alpha \) true, thus making \( B \) as a whole trivially true.

So far so good... but we know that something must still be wrong with this proof, somehow, since the algorithm is after all incorrect: our sequence of arrows \( \rightarrow, \Rightarrow, \Rightarrow \) is a counterexample.\(^7\) But where is the error?

\(^7\)A proof must be incorrect, no matter how elegant, if it proves a falsehood. An interesting question (though not related to this example), is how to discredit convincingly an incorrect proof of something that is true.
• The bump in the carpet

The problem that we have moved (not re-moved) is of course that Assertion $B$ is not globally correct after all — it can be invalidated by execution of the `await`-statement labelled $\alpha$ in Process $A$ of Fig. 3, since that execution makes $\alpha$ false and thus places $B$ at risk.

We solve this in Fig. 4 by adding at Label $\alpha$ an `await`-statement that cannot be executed -- cannot falsify $\alpha$ -- while control in $Q$ is at (the new) Label $\beta$, the location of Assertion $B$ whose global correctness we are trying to establish. It is, for the moment, expressed in terms of a location assertion $\beta$ — but we will soon fix that by adding an extra variable.

• Finally, the first (partial) solution

By adding a new variable $b_Q$ to $Q$ with $\neg b_Q \Rightarrow \neg \beta$ invariant, we can in Fig. 5 code the guard of our new `await` in $P$ while (mildly) strengthening it.\(^{10}\)

Note that its six (blue) assertions can be checked for local- and for global correctness (thus $6\times2 = 12$ separate checks) in any order and without having in one check to remember what you were thinking about while you did some other.

This is a crucial advantage of the Owicki-Gries method over operational reasoning. Fig. 6 sets out the details: there are surprisingly few.

• The complete solution

With the globally correct assertions of Fig. 5 we have what we need for the proofs of safety and liveness.

If we apply our earlier reasoning analogously to the interactions between the other process pairs, we get the code of Fig. 7.

• Liveness

We now give a sketch of the liveness proof for the algorithm of Fig. 7. First we prove absence of deadlock.

For deadlock, every process must be either at a blocking `await` or in its $OB$, and there must be at least one of the former. Observe first (from the annotations) that therefore for all processes $X$ we have $\neg b_X$, and hence all blocking `await`s must be of the form $x<y \vee y=0$ where $<$ is either $<$ or $\leq$.

(That is, no blocking `await` $\neg b_X$ can be involved in a deadlock.)

Write $X\sim Y$ when process $X$ is blocked at `await` $x<y \vee y=0$.

Since all processes $Y$ in $OB$ have $y=0$, if $X\sim Y$ then $Y$ cannot be in $OB$. Rather we must have $Y\sim Z$ in turn, where $Z$ could be $X$ but is not necessarily.

Yet still there must be a loop $Y\sim \cdots \sim Z\sim \cdots \sim Y$ for $Y,Z$ somewhere, since there are only finitely many processes. Since $V\sim W$ implies $w\neq 0$ for any pro-

---

8The impossibility of getting rid of a bump in the carpet is yet another example of invariant-based reasoning in everyday life. (Remember the dish-washing machine.\(^{9}\)) In this case the invariant is that the area of the carpet is bigger than the area of the room, an invariant preserved by all attempts to push the bump around.

9We did not discuss the dish-washing machine (its invariant) in 2021. Ask me about it.

10One observes, though, that $q\neq 0 \Rightarrow \neg \beta$ is invariant already. Why not forget $b_Q$ and simply introduce `await` $q\neq 0$ instead?
$p = q = r = 0$

\[
\begin{align*}
OB & \quad OB & \quad OB \\
q_p := q & \quad r_Q := r & \quad p_R := p \\
r_p := r & \quad p_Q := p & \quad q_r := q \\
p := 1 + q_p \lor r_p \ \{p \neq 0\} & \quad \{b_Q \} \ \{p \leq p_q \lor \alpha\} & \quad q := 1 + r_Q \lor p_Q \\
& \quad b_Q := F & \quad r := 1 + p_R \lor q_R \\
\alpha: & \quad \text{await } \neg b_Q & \quad \text{await } q \leq r \lor r = 0 & \quad \text{await } r < p \lor p = 0 \\
\{\neg \alpha\} \ \{p \leq q \lor q = 0\} & \quad \text{await } p \leq r \lor r = 0 & \quad \text{await } q < p \lor p = 0 & \quad \text{await } r < q \lor q = 0 \\
\{p \neq 0\} \ \{\neg \alpha\} \ \{p \leq q \lor q = 0\} & \quad \{q \neq 0\} \ \{q < p \lor p = 0\} \\
CS & \quad CS & \quad CS \\
p := 0 & \quad q := 0 & \quad r := 0 \\
\end{align*}
\]

We add an extra variable $b_Q$ to $Q$ so that we can code the new `await` in $P$ without referring to a label.

This handles only $Q$'s potential interference with $P$'s assertion $p \leq q \lor q = 0$. In Fig. 7, analogous coding is added for the other five interactions.

The red assertion at $Q$'s critical section is (implied by) what we could establish with similar reasoning between $Q$ and $P$, i.e. complementary to what we have done above. It is inconsistent with the assertion at $P$'s critical section, thus establishing safety between $P$ and $Q$. By symmetry, similar arguments will work for the two other cases.

Figure 5: Correct code, with assertions in blue, for managing the interference of Process $Q$ with Process $P$ in the three-way Bakery Algorithm.
### Assertions are checked top-to-bottom, left-to-right

<table>
<thead>
<tr>
<th>Assertion</th>
<th>Local correctness</th>
<th>Global correctness</th>
</tr>
</thead>
<tbody>
<tr>
<td>{p \neq 0}</td>
<td>(q_p, r_P \geq 0)</td>
<td>(p) is owned by (P)</td>
</tr>
<tr>
<td>{¬α} {p \leq q \vee q = 0}</td>
<td>property</td>
<td>(q:= 0) trivially non-interfering;</td>
</tr>
<tr>
<td>of await</td>
<td>{¬α} {p \leq p_Q \vee α}</td>
<td>(q:= 1 + r_Q \cup p_Q) {p \leq q}</td>
</tr>
<tr>
<td>{p \neq 0}</td>
<td>carried through from above</td>
<td></td>
</tr>
<tr>
<td>{¬α} {p \leq q \vee q = 0}</td>
<td>carried through from above</td>
<td></td>
</tr>
<tr>
<td>{b_Q}</td>
<td>trivial</td>
<td>(b_Q) is owned by (Q)</td>
</tr>
<tr>
<td>{b_Q}</td>
<td>carried through from above</td>
<td></td>
</tr>
<tr>
<td>{p \leq p_Q \vee α}</td>
<td>trivial</td>
<td>(p:= 0) non-interfering because (p_Q \geq 0); (p:= 1 + q_P \cup r_P) non-interfering because it establishes (α); \textbf{await} (¬b_Q), which would falsify (α), cannot execute due to precondition {b_Q}</td>
</tr>
<tr>
<td>{q \neq 0}</td>
<td>(q &lt; p \vee p = 0)</td>
<td>by analogous reasoning wrt. interference of (P) with (Q)</td>
</tr>
</tbody>
</table>

The \textit{owns} property, identified by Lamport, obtains when only one process may write to a variable although others may read it: the writing process is said to own the variable.

All integer variables are non-negative at point of use, invariant for \(p, q, r\) and trivially established for \(\{p, q, r\}_{\{P, Q, R\}}\) by local reasoning.

The local- and global correctness of label-assertions are taken as self-evident.

Other assertions are in bold at the point of introduction, where their local correctness is first established; a justification (elsewhere) of “carried through” implies a check that they have not been falsified by local commands, or interfered with by global commands, since their introduction.

Figure 6: Checking assertions in Fig. 5.
\[ p = q = r = 0 \]

\[ O \]

\begin{align*}
\text{b}_P &:= \top \\
\text{q}_P &:= q \\
\text{r}_P &:= r \\
\text{p} &:= 1 + \text{q}_P \land \text{r}_P \\
\text{b}_P &:= F \\
\text{q}_P &:= 1 + \text{r}_P \land \text{q}_P \\
\text{b}_P &:= F \\
\text{q}_P &:= \text{r}_P \\
\text{r}_P &:= \text{p}_P \\
\end{align*}

\[ \text{OB} \]

\begin{align*}
\text{b}_Q &:= \top \\
\text{r}_Q &:= r \\
\text{p}_Q &:= p \\
\text{q}_Q &:= 1 + \text{r}_Q \land \text{p}_Q \\
\text{b}_Q &:= F \\
\text{r}_Q &:= 1 + \text{p}_Q \land \text{q}_Q \\
\text{b}_Q &:= F \\
\text{r}_Q &:= \text{p}_Q \\
\text{p}_Q &:= \text{q}_Q \\
\text{q}_Q &:= \text{p}_Q \\
\text{r}_Q &:= \text{q}_Q \\
\end{align*}

Analogous coding has been added to Fig. 5 for the other five interactions; the assertions are removed for clarity. If we returned to our original story, it would now run approximately as follows.

Every customer has a card, either blank or with a positive number written on it. A customer’s card is initially blank; when he enters the bakery it is blank; and when he leaves the bakery, he erases it so that it is blank again.

After he enters the bakery, he sees all the customers standing in order of height against the wall; he joins them, at his proper position. He then leans forward and looks at all their cards in order, from the shortest person to the tallest; when he has finished, he writes on his own card the number one larger than the largest he saw. (If he saw none, because all cards were blank, then he writes 1.)

He then leans forward again (a second sweep) and looks at the cards, again from shortest to tallest in order. If a person’s card is blank, he waits until it is non-blank; if a shorter person’s card is strictly less than his own, he waits until it is equal or greater; if a larger person’s card is equal or less than his own, he waits until it is strictly greater.

Once he has looked at them all, he walks to the counter.

This story can probably be improved: \textit{can you write a better one?}

Figure 7: Correct code for all interactions in the three-way Bakery Algorithm.
cesses $V, W$, from the loop above we have a corresponding loop $y < z < \cdots < y$ of inequalities, at least one of which must be strict. And that is impossible.

The other liveness property we must prove is absence of starvation, i.e. that there is no possibility one (or more) processes make no progress while others continue to circulate. Although this might at first seem to be a special case of the above, in fact it is not: the reason is as follows.

If there are circulating processes (at least one), then they might intermittently clear the blocking conditions of the others, so that those conditions never hold all at the same time (as we assumed they did above). Then the adversarial scheduler, “in league” with the circulating processes, can arrange always to activate the starving processes only when they are blocked.\(^{11}\)

We can handle this more complicated situation by a slightly informal argument, one showing that if even just a single process is starving then eventually overall deadlock will result. This deadlock we have already shown to be impossible, so that starvation will also have been shown to be impossible,\(^{12}\)

If a process $X$ is starving, then $x \neq 0$ is continuously true, as we saw above. And if some other process $Y$ is forever circulating, then eventually that $Y$ will execute its statements

$$xy := x \quad \cdots \quad y := 1 + (\cdots \sqcup xy \sqcup \cdots) \quad \cdots \quad \text{await } y < x \lor x = 0.$$  

Moreover, it will encounter no interference from $X$, since $X$ is starving (and thus blocked). That being so, we can informally annotate $Y$’s code above with\(^{13}\)

\[
\begin{align*}
xy := x & \quad \{xy=x\} \\
\{xy=x\} & \quad y := 1 + (\cdots \sqcup xy \sqcup \cdots) \quad \{y>x\} \\
\{y>x \land x \neq 0\} & \quad \text{await } y < x \lor x = 0 ,
\end{align*}
\]

and see that $Y$ becomes blocked at its \texttt{await} — so that it in fact is not forever circulating, contradicting our assumption that it was doing so.

Thus if some $X$ is starving, then there can be no (other) forever circulating process $Y$ — i.e. all other processes eventually must either block, or reach their $OB$ and remain there. But those deadlocks we have already excluded.

- **Atomicity**

Unlike any previous algorithm, and almost all subsequent algorithms, the bakery algorithm works regardless of what value is obtained by a read that overlaps a write. I don’t know how many people realize how remarkable [that] is... [and] I don’t know when [others] finally reconciled [themselves] to the algorithm’s correctness.

Several books have included emasculated versions of the algorithm in which reading and writing are atomic operations, and called those versions “the bakery algorithm”. I find that deplorable.

---

\(^{11}\)Recall the earlier comment about the “adversarial child” who rings her parents every day, but makes sure it is never when they are at home.

\(^{12}\)Note that e.g. $P$ cannot starve $Q$ even though it wins the tie whenever $p=q$!

\(^{13}\)Why must we consider these annotations to be informal?
As Lamport points out in his original article on the Bakery, what we seem to have done so far is not so much to “solve” the mutual-exclusion problem as to move it: we have simply reduced it to another mutual exclusion that itself needs to be implemented somewhere else.

What Lamport means is that we have so far assumed that individual “atomic” statements in each process are executed one at a time, without overlapping between processes even though the statements can be arbitrarily interleaved between processes. Some thought shows this “statement by statement” atomicity assumption to be equivalent to enforcing atomicity of the read- and write actions within the statements: those actions refer to the global memory containing our shared variables (here \( p, q, r \) and \( b_{P,Q,R} \)), and they execute separately one after the other.\(^{14}\) That is the sense in which the mutual exclusion has only been moved: it is now at that lower level.

It turns out that Lamport’s algorithm does not need that atomicity assumption, though it is not obvious that it doesn’t (and it was not at first obvious to Lamport,\(^{1}\) either). Instead, references to the global memory may overlap, meaning that the same shared variable (say \( p \)) may be written in one process (say the assignment \( p := 0 \) in \( P \)) while it is being read in another (say \( p_Q := p \) in \( Q \)). In that case the value read (by \( Q \)) could be anything at all.\(^{15}\)

We note however that the Bakery Algorithm never writes simultaneously to a global variable, whereas Peterson’s does. Lamport identified the “ownership” of a shared, global variable by the sole process that writes to it.\(^{1}\)

And the Bakery Algorithm works even so. To show that, in Fig. 8 we add two statements \( p := \text{anything} \), and two new occurrences of the label \( \alpha \): recall that a label, as an assertion, holds when \( \text{any} \) of its occurrences is about to execute, so that \( \alpha \) now holds at three places. This new, slightly altered code is designed so that any overlapping of references to \( p \) can be seen as interleaved references to \( p \)—for which we have correctness already.\(^{16}\) That is, the earlier code of Fig. 7 works with overlapping because the code of Fig. 8 works with interleaving.\(^{17}\)

One should have a healthy suspicion of claims like this however, and it’s very important to spend time looking for “what if” scenarios that might disprove them. Here are some for the current case, referring to Fig. 7 of course since that is the actual algorithm that’s supposed to be correct even with overlapping:

1. Suppose that just as Process \( P \) is reading \( q \) via the assignment \( q_P := q \), Process \( Q \) interferes by writing to \( q \) with \( q_Q := 0 \).

Then \( q_P \) in \( P \) could receive an arbitrary value, for example \(-1\), so that the

---

\(^{14}\)This sometimes implicit assumption occasionally surfaces when we note that our individual statements should refer to a single global variable at most: for then (effective) atomicity of the statements is guaranteed by atomicity of the reads/writes within them. For that reason we must sometimes rewrite what we want, i.e. \( p := 1 + (q \lor r) \), in terms of what atomicity allows us to get, i.e. \( p := 1 + (q_P \lor q_{P,Q,R}) \) with the preceding, separate assignments to \( q_P \) and \( r_{P,R} \).

\(^{15}\)In the examples we gave, the value read by \( p_Q := p \) in \( Q \) need not be the value 0 that Process \( P \) is currently writing via \( p := 0 \), nor the value of \( p \) just before \( P \) began writing that 0, nor even a value that \( P \) wrote at any other time.

Without an assumption of atomicity at \( \text{any} \) level, we could e.g. have \( P \) writing \( p \)'s bits from left to right while at the same time \( Q \) is reading them from right to left.

\(^{16}\)The \( b \)-variables can be treated similarly.

\(^{17}\)There is a small assumption being made here, that such overlapping occurs at most once per write. In general that is not true: for example, if both Processes \( Q, R \) read \( p \) while \( P \) was writing it, then in our approximation they would receive the same (corrupted) value of \( p \). In reality it could be worse.
\[ p = q = r = 0 \]

\[ OB \]

\[ q_p := q \]
\[ r_p := r \]
\[ p := \text{anything} \]
\[ \alpha : p := 1 + q_p \lor r_p \quad \{ p \neq 0 \} \]
\[ \alpha : p := 0 \]
\[ q := 0 \]
\[ r := 0 \]

We add the “anything” assignments to \( p \) (and correspondingly to \( q, r \) but not shown here), together with two new occurrences of Label \( \alpha \). In effect \( \alpha \) means, for other processes, that variable \( p \) has been interfered with in an unpredictable way. Obviously the local correctness of the assertions is unaffected. Remarkably, their global correctness is not affected either.

Figure 8: Justifying the overlap of reads and writes.

subsequent assignment \( p := 1 + q_p \land r_p \) in \( Q \) would assign 0 to \( p \), making it seem e.g. to \( Q, R \) that Process \( P \) is safely in its \( OB \) — whereas in fact it is bearing down on its \( CS \) and indeed will successfully enter it since \( p = 0 \), and thus \( p \leq q, r \) no matter what \( Q, R \) might be doing.

The scenario above suggests we have relied in our proof, somewhere, on our variables’ all being non-negative. (Indeed we have.) But where?

This experiment has alerted us to an issue with overlapping: if a random value \emph{outside} the expected type is returned from a read, we must reassign to it some value \emph{within} its type. It doesn’t matter which. (It’s an issue also with our \( b \) Booleans, encoded say in \( C \)-style as 0,1 — then an overlapped read could return “the Boolean 2.” We have to check for that, and replace it either by 0 or 1 arbitrarily.)

Our added statements in Fig. 8 should be \( p := \text{any nonnegative integer} \).

2. \textit{Suppose that just as \( P \) reads \( q \) in its \texttt{await} \( p \leq q \lor q = 0 \), Process \( Q \) interferes by writing to \( q \) with \( q := 1 + r_Q \land p_Q \).}

Then \( P \) could receive a high value for \( q \), or simply 0, either way satisfying \( p \leq q \lor q = 0 \) and so allowing \( P \) to proceed towards \( CS \). What if the actual
value assigned to $q$ by $Q$ was nonzero but lower than $p$, so that instead $P$ should have waited?

Rebuttal: If $q$'s new value is actually lower than $p$, then $p_Q$ must be lower than $p$ as well. If that is so then $P$ must have assigned to $p$ between $Q$'s earlier execution of $p_Q := p$ and now, as $Q$ executes $q := 1 + r_Q \sqcup p_Q$, and during that whole period $b_Q$ was true. But while $b_Q$ is true, Process $P$ cannot pass from its assignment to $p$ through its `await `p\leq q \vee q=0 we are assuming is its current position.  

3. But surely... if any read of $q$ by Processes $P$ can return an arbitrary value if $Q$ just happens to be scheduled to write to $q$ at the same moment, our assumption of adversarial scheduling means we can assume that all $P$'s reads of $q$ are interfered with in that way, i.e. that $P$ receives an arbitrary value for $q$ on every occasion. In that case, we can simply replace $P$'s reads of $q$ by locally determined random assignments — and then $P$ is no longer affected by $Q$ in any way. In that case, they cannot possibly respect mutual exclusion for $CS$.

Rebuttal: Although every read of $q$ by $P$ is vulnerable to interference by $Q$, in fact it is not possible to arrange for that to happen every time on a single run: some of those reads will succeed without interference, no matter how hard the scheduler tries to arrange otherwise. Thus $P$ and $Q$ cannot be decoupled via arbitrary assignments in the way suggested.

Let’s be the adversarial scheduler ourselves, and try to arrange that $P$’s every read of $q$ is corrupted by $Q$’s writing to $q$ at the same moment.

(a) We assume $R$ does not participate, in which case $r=0$ is invariant, and assignments to $r_P$, $r_Q$ will always be 0. Then because both $(\leq 0)$ and $(\vee 0 = 0)$ are identities, we can simply leave all references to $r$ out of the following.

(b) $P$’s first read of $q$ is its $q_P := q$; we schedule $Q$ so that it executes its $q := 1 + p_Q$ at the same moment. (We have left $r_Q$ out, as justified in Step 3a.) Since $p_Q$ will be 0 the assignment sets $q$ to 1. But let’s assume that $P$ reads (the corrupted) value 20 into $q_P$.

(c) $P$’s next read of $q$ is at the `await `p\leq q \vee q=0, but on the way it will have set $p$ to $1+q_P$, that is to 21.

(d) In order to interfere with $P$’s read of $q$ at the `await` statement, we schedule $Q$ so that it executes $q := 0$ at that moment, just leaving its critical section. (It will not have been held up at its `await`, since $q < p$ evaluates to 1<21, i.e. to $T$.)

(e) Although it is tempting to deliver a corrupted “high” value for $q$ to $P$ at this point, to make the test $p \leq q$ give the wrong answer, we notice that once $q := 0$ has executed, the `await` will not block $P$ in any case; thus we decide to deliver value $q=10$ to $P$ so that it evaluates 21\leq10, and thus waits when it needn’t have.

(f) The next time $P$ reads $q$ will be when it retries the `await` condition; to interfere with that again, we schedule $Q$ so that it is executing $q := 1 + p_Q$ at the same moment. Since $p_Q$ will have been set to $p$, that is to 21, the new value of $q$ will be 22. But again we deliver the corrupted $q=10$, so that $P$ waits unnecessarily a second time.

---

18 Try to read that paragraph aloud while sitting on your hands.
(g) The next time $P$ reads $q$ will again be the await, its third attempt; and for $Q$ to interfere with that it must reach its next assignment $q:=0$ to $q$, for which it must pass through its await $q<p \lor p=0$.

(h) But it cannot do that: since $P$ is stuck, it cannot corrupt $Q$’s read of $p$, and so the correct comparison, that is $22<21 \lor 21=0$ will occur at await $q<p \lor p=0$ — and $Q$ will be blocked.

(i) Thus the next read of $q$ by $P$, at its await, will be interference-free, our rebuttal of the claim that it’s easy to arrange that all of $P$’s reads of $q$ are corrupted by $Q$.

(j) The result will be that $p\leq q$ is $21\leq 22$, and $P$ will pass through its await, now on the way to its critical section — at last.

It is truly astonishing that Lamport’s algorithm is robust against corruption of this kind, almost magical. It’s the magic of careful (in-)formal, non-operational reasoning, effort that –once spent– pays off forever.

 Remember though — the above rebuttals are not correctness arguments. They are common-sense checks we carry out on our own reasoning, to make it more certain that we haven’t made a mistake. Hand-waving like the above should never be offered, nor accepted as a proof of correctness.

• Epilogue

In spite of the paragraph just above, the three “what if” scenarios –particularly the second one— are typical examples of correctness reasoning about programming as it is currently practised in general. Once “sufficiently many” successful scenarios are walked through, the software is considered to be correct — or, at least, correct enough to be released.

If the software’s correctness is important (but see below), then this is completely the wrong approach.

Correctness should be established by careful analysis using sound methods that help us to avoid mistakes in our reasoning. After that, scenario-explorations like the above have their place as a second, supplementary check that we have not made a faulty deduction, missing an error whose effects might appear only years later and once the system is distributed world-wide. But such experiments, no matter how many, must never be the primary or the only assurance we have for the quality of our work.

Analysis is expensive, however. Earlier we studied Peterson’s algorithm for two-process mutual exclusion, just four lines long with its two initial assignments, an await and then a final assignment. Dekker’s original \texttt{MutEx 1.0} was upgraded, after 15 years, to Peterson’s \texttt{MutEx 2.0} that was thus programmed at the rate of $0.004/(15 \times 365)$, that is less than $10^{-6}$ kloc/day. Put inversely, Peterson’s productivity was roughly 1 kilo-day per line of code.

From Wikipedia:

There were several early failed attempts at proving the \textit{Four-Colour Theorem}. One proof was given by Alfred Kempe in 1879, which was widely acclaimed; another was given by Peter Guthrie Tait in 1880. It was not until 1890 that Kempe’s proof was shown incorrect by Percy Heawood, and in 1891 Tait’s proof was shown incorrect by Julius Petersen — each false proof stood unchallenged for 11 years.

It took 124 years to find the first proof (1976) that is still standing today (2021).
Given such low productivity, is it worth being so careful?

Indeed, correctness does not always matter. Suppose you had an integer-factorisation algorithm that ran in linear time, but occasionally was wrong (and so cannot have been proved correct). Would you use it? Of course! You would apply that unreliable algorithm to internet public-key moduli and you would check the factors by multiplying back. If they’re wrong, you’ve lost a little; but if they’re right, you’ve gained a lot (though possibly a jail term as well).

Another context is in the commercial world where the code is required yesterday: there is no time for a proof. (And in many cases the code that was “so urgently required” won’t actually be used until next year, if at all; so its correctness is irrelevant.)

Still another correctness-insensitive context is “user friendliness” where in fact Unix-based command interfaces are the most logical, and most easily reasoned about, but in general they are not what computer users want. More important for them is that it should be obvious what to click next and, hopefully, how to “undo” if the result is unexpected.

Of course sometimes correctness does matter; and it is up to you to recognise those situations, to decide when that is, to be able to spend the time and effort where it is needed (because you haven’t wasted it where it was not needed). Yet there will be little point unless you actually know how to use rigorous techniques when indeed they are required. Otherwise you’ll just be telling your boss that your latest job needs to be done by someone else, someone more expert than you are.

This short, introductory course is not teaching you any formal method in depth; but it is trying to show you what Formal Methods are, and thus to help you make up your own mind whether further study in one or more of them is something you’d like to put in your career path. As a spinoff, however, it might be giving you you some insight into how to think more effectively about everyday programming as well.

Carroll Morgan
July 2021