Bounds for Mutual Exclusion with only Processor Consistency

Lisa Higham* and Jalal Kawash†
Department of Computer Science, The University of Calgary, Canada, T2N 1N4
Fax: +1 (403) 284 4707, Phone: +1 (403) 220 7696, 220 7681
{higham|kawash}@cpsc.ucalgary.ca

Abstract

Most weak memory consistency models are incapable of supporting a solution to mutual exclusion using only read and write operations. Processor Consistency–Goodman's version is an exception. Ahamad et al.[1] showed that Peterson's mutual exclusion algorithm is correct for PC-G, but Lamport's bakery algorithm is not. In this paper, we derive a lower bound on the number and type (single- or multi-writer) of variables that a mutual exclusion algorithm must use in order to be correct for PC-G. We show that any such solution for n processes must use at least one multi-writer and n single-writers. This lower bound is tight when $n = 2$, and is tight when $n \geq 2$ for solutions that do not provide fairness. We show that Burns' algorithm is an unfair solution for mutual exclusion in PC-G that achieves our bound. However, five other known algorithms that use the same number and type of variables are incorrect for PC-G. A corollary of this investigation is that, in contrast to Sequential Consistency, multi-writers cannot be implemented from single-writers in PC-G.

1 Introduction

The Mutual Exclusion Problem is the most famous and well-studied problem in concurrency. Following Silberschatz et al.[13], we refer to this problem as the Critical Section Problem (CSP) to distinguish the problem from the Mutual Exclusion Property. In CSP, a set of processes coordinate to share a resource, while ensuring that no two access the resource concurrently. CSP solutions for Sequential Consistent memory have been known since the 1960s; Raynal [12] provides an extensive survey. In fact, as shown by Lamport [9], even single-reader single-writer bits suffice to solve the critical section problem, as long as accesses to these seemingly weak objects are guaranteed to be Sequentially Consistent.

Most weak memory consistency models, however, are incapable of supporting a solution to mutual exclusion using only read and write operations [6]. An exception is Processor Consistency (abbreviated PC-G)[1] as proposed by Goodman and formalized by Ahamad et al.[1], who show that Peterson's mutual exclusion algorithm [11] is correct for PC-G. However, Lamport's bakery algorithm has been shown to fail for PC-G [1]. We are thus motivated to determine what is necessary and sufficient to solve CSP with only Processor Consistent Memory with only reads and writes to shared variables. For example, Peterson's algorithm makes use of multi-writers, variables that can be written my more than one process, while Lamport's

*Supported in part by the Natural Sciences and Engineering Research Council of Canada grant OGP0041900.
†Supported in part by a Natural Sciences and Engineering Research Council of Canada doctoral scholarship and an Izaak Walton Killam Memorial scholarship.
[1] Several variants of Processor Consistency exist. The one referred to in this paper is due to Ahamad et al.'s[1] interpretation of Goodman's original work [4].

bakery algorithm [7] uses only single-writers, variables that can be written by exactly one process. Are multi-writers essential?

In this paper, we derive bounds on the number and type (single- or multi-writer) of variables that a mutual exclusion algorithm must use in order to be correct for PC-G. Specifically, any PC-G solution for n processes must use at least one multi-writer and n single-writers. We prove that Burns' algorithm [3], which uses one multi-writer and n single-writers, is an unfair solution for mutual exclusion in PC-G. Thus our bound is tight for unfair solutions to CSP. Since Peterson's 2-processor algorithm is fair and correct for PC-G, our bound is tight even for fair solutions when $n = 2$.

We further investigate properties that a solution, using one multi-writer and n single-writers, must satisfy in order to be correct for PC-G. Using these properties, we establish that five algorithms [12], Dekker's, Dijkstra's, Knuth's, De Bruijn's, Eisenberg and MacGuire's, are incorrect for PC-G. All of these have been developed for Sequential Consistency (SC) [8], and all use one multi-writer and n single-writers. However, most of these algorithms are fair solutions for CSP in SC. The only fair solution we have found for PC-G is Peterson's which uses $n-1$ multi-writers and n single-writers.

Since multi-writers are required to solve CSP in PC-G, a corollary of our investigation is that, in contrast to Sequential Consistency, multi-writers cannot be implemented from single-writers in PC-G.

The PC-G model is defined in Section 2; and CSP is defined in Section 3. Section 4 provides a template for our impossibility proofs, which is used to establish our lower bounds in Section 5. The major results in Section 5 have been automatically verified using the SPIN model checker [5].

2 The Model

A multiprocess system can be modeled as a collection of processes operating on a collection of shared data objects. For this paper, the shared data objects are variables supporting only read and write operations, where $r(x)v$ and $w(x)v$ denote, respectively, a read operation of variable x returning v and a write operation to x of value v. An operation can be decomposed into invocation (performed by processes) and response (returned by variables) components.

It suffices to model a *process* as a sequence of read and write invocations, and a *multiprocess system* as a collection of processes together with the shared variables. Henceforth, we denote a multiprocess system by the pair (P,J) where P is a set of processes and J is a set of variables. A *process computation* is the sequence of reads and writes obtained by augmenting each read invocation in the process with its matching response. A *(multiprocess) system computation* is a collection of process computations, one for each process in the collection. Let O be all the (read and write) operations in a computation of a system (P,J). Then, $O|p$ denotes all the operations that are in the process computation of process $p \in P$; $O|x$ denotes all the operations that are applied to variable $x \in J$, and $O|w$ denotes all the write operations.

A sequence of read and write operations to variable x is *valid* if and only if each read in the sequence returns the value of the most recently preceding write. Given any collection of read and write operations O on a set of variables J, a *linearization of* O is a (strict) linear order[2] $(O, \xrightarrow{L})$ such that for each variable x in J, the subsequence $(O|x, \xrightarrow{L})$ of $(O, \xrightarrow{L})$ is valid.

Let O be a set of operations in a computation of a system (P,J). Define the *program order*, denoted $(O, \xrightarrow{prog})$, by $o_1 \xrightarrow{prog} o_2$ if and only if o_2 follows o_1 in the computation of p.

A *(memory) consistency model* is a set of constraints on system computations. A system (P,J) *satisfies* memory consistency D if every computation that can arise from it meets all the constraints in D. Three

[2]A *(strict) partial order* (simply, partial order) is an anti-reflexive, transitive relation. Denote a partial order by a pair (S,R). The notation $s_1 R s_2$ means $(s_1, s_2) \in R$. A *(strict) linear order* is a partial order (S,R) such that $\forall x,y \in S\ x \neq y$, either xRy or yRx.

memory consistency models are defined here: Sequential Consistency (SC) [8], Pipelined-Random Access Machine (P-RAM)[10], and PC-G [1].

Definition 2.1 *Let O be all the operations of a computation C of a multiprocess system (P,J). Then C satisfies* SC *if there is a linearization $(O, \xrightarrow{L})$ such that $(O, \xrightarrow{prog}) \subseteq (O, \xrightarrow{L})$.*

Definition 2.2 *Let O be all the operations of a computation C of a multiprocess system (P,J). Then C satisfies* P-RAM *if for each process $p \in P$ there is a linearization $(O|p \cup O|w, \xrightarrow{L_p})$ such that $(O|p \cup O|w, \xrightarrow{prog}) \subseteq (O|p \cup O|w, \xrightarrow{L_p})$.*

Definition 2.3 *Let O be all the operations of a computation C of a multiprocess system (P,J). Then C satisfies* PC-G *if for each process $p \in P$ there is a linearization $(O|p \cup O|w, \xrightarrow{L_p})$ such that*

1. *$(O|p \cup O|w, \xrightarrow{prog}) \subseteq (O|p \cup O|w, \xrightarrow{L_p})$, and*
2. *$\forall q \in P$ and $\forall x \in J$, $(O|w|x, \xrightarrow{L_p}) = (O|w|x, \xrightarrow{L_q})$.*

Let A and D be an algorithm and a memory consistency model, respectively. Then, *A solves CSP for D* if for every system S that satisfies D, A solves every instance of CSP on S.

3 Critical Section Problem

We denote a CSP problem by CSP(n) where n is the number of processes in the system. Each process has the following structure:

repeat
<*remainder*>
<*entry*>
<*critical section*>
<*exit*>
until *false*

A solution to CSP(n), $n \geq 2$, must satisfy the following properties[3]:

- **Mutual Exclusion:** At any time there is at most one process in its <*critical section*>.
- **Progress:** If at least one process is in <*entry*>, then eventually one will be in <*critical section*>.
- **Fairness:** If a process p is in <*entry*>, then p will eventually be in <*critical section*>.

[3]Other forms of defining solution properties are possible as is given by Attiya et al.[2].

4 Template for Impossibility and Lower Bound Proofs

We will use the partial computations 1, 2, and 3 defined below. First, assume for the sake of contradiction that there exists an algorithm A that solves CSP(n) for a given memory consistency model, D, for $n \geq 2$. This solution must work when exactly two processes, say p and q, are participating and the rest engaging in <*remainder*>. If A runs with p in <*entry*> and with q in <*remainder*>, then by the Progress property, p must enter its <*critical section*> producing a partial computation of the form of Computation 1, where λ denotes the empty sequence and o_i^p denotes the i^{th} operation of p.

Computation 1 $\begin{cases} p : o_1^p, o_2^p, ..., o_k^p \;\; (p \textit{ is in its } < \textit{critical section} >) \\ q : \lambda \end{cases}$

Similarly, if A runs with q's participation only, Progress guarantees that Computation 2 exists.

Computation 2 $\begin{cases} p : \lambda \\ q : o_1^q, o_2^q, ..., o_l^q \;\; (q \textit{ is in its } < \textit{critical section} >) \end{cases}$

Now, consider Computation 3 where both p and q are participating, but both are in their <*critical section*>. By assumption, both computations 1 and 2 both satisfy D. If we can show that Computation 3 also satisfies memory consistency condition D, the desired contradiction is achieved, since mutual exclusion is violated. This implies that A does not exist.

Computation 3 $\begin{cases} p : o_1^p, o_2^p, ..., o_k^p \;\; (p \textit{ is in its } < \textit{critical section} >) \\ q : o_1^q, o_2^q, ..., o_l^q \;\; (q \textit{ is in its } < \textit{critical section} >) \end{cases}$

None of the arguments in the following theorems depends on the Fairness property, so the impossibilities include unfair solutions as well. Furthermore, none of these argument depends on the size of variables. So, these results apply to unbounded variables as well.

5 Bounds on CSP for PC-G

Ahamad et al.[1] proved that Peterson's algorithm [11], which was originally developed for SC systems, solves CSP(2) for PC-G. Given algorithm A_2 that solves CSP(2) for PC-G an algorithm A_n that solves CSP(n) for PC-G, where $n \geq 2$, can be constructed from A_2 by building a tournament tree. Processes are partitioned into sets of size two each. For each set, A_2 is used to select a "winner". The winners are again partitioned into sets of size two, and A_2 can be used in this manner repeatedly until only one winner remains. Thus we conclude that there is an algorithm that solves CSP(n) for PC-G.

This section further investigates bounds and restrictions on these PC-G solutions.

5.1 Type of Variables

A *multi-writer* variable (simply, multi-writer) can be updated by any number of processes in the system, while a *single-writer* variable (simply, single-writer) can be updated by exactly one designated process.

We show that the use of multi-writers is crucial to solve CSP on PC-G. First we need the following lemma.

Lemma 5.1 *In a system* (P,J) *where* J *consists entirely of single-writers, PC-G is equivalent to P-RAM.*

Proof: Obviously, PC-G is at least as strong as P-RAM. We show that without the use of multi-writer variables, P-RAM is at least as strong as PC-G. Let $(O|p \cup O|w, \xrightarrow{L_p})$ and $(O|q \cup O|w, \xrightarrow{L_q})$ be linearizations for p and $q \in P$ that are guaranteed by P-RAM. Since, for any variable $x \in J$, there is only one process, say s, that writes to x, and both $(O|p \cup O|w, \xrightarrow{L_p})$ and $(O|q \cup O|w, \xrightarrow{L_q})$ have all these writes to x in the program order of s, the order of the writes to x in $(O|p \cup O|w, \xrightarrow{L_p})$ is the same as the order of the writes to x in $(O|q \cup O|w, \xrightarrow{L_q})$. Therefore, the definition of PC-G (Definition 2.3) is satisfied. ■

CSP, however, is impossible for P-RAM:

Theorem 5.2 *There does not exist an algorithm that solves CSP(n) for P-RAM, even if* $n = 2$.

Proof: Assume that there is an algorithm A that solves CSP(n) for P-RAM. Then computations 1 and 2 exist. Define the following sequences for p and q, respectively, for Computation 3.

$$(O|p \cup O|w, \xrightarrow{L_p}) = \langle (o_1^p, \ldots, o_k^p), (o_1^q, \ldots, o_l^q)|w \rangle$$

$$(O|q \cup O|w, \xrightarrow{L_q}) = \langle (o_1^q, \ldots, o_l^q), (o_1^p, \ldots, o_k^p)|w \rangle$$

Clearly, each preserves $\xrightarrow{prog}$ as required by the definition of P-RAM. Also, each is a linearization because the first part (for instance, $(o_1^p, \ldots, o_k^p)$) corresponds to a possible computation, and the second part (for instance, $(o_1^q, \ldots, o_l^q)|w$) contains only writes. Thus, Computation 3 is P-RAM. Therefore, our assumption must have been in error and A does not exist. ■

Theorem 5.3 *There does not exist an algorithm that uses only single-writers and solves CSP(n) for PC-G, even if* $n = 2$.

Proof: This follows immediately from Lemma 5.1 and Theorem 5.2. ■

Ahamad et al.[1] also prove that Lamport's Bakery algorithm [7], which uses only single-writers, is incorrect for PC-G. The consequence of Theorem 5.3 is that any CSP solution for PC-G must use at least one multi-writer.

Multi-writer variables can be constructed from single writer variables in a SC memory system[14]. However, this is not the case in PC-G.

Corollary 5.4 *Multi-writers cannot be implemented from single-writers in PC-G memory system.*

Proof: Peterson's algorithm solves CSP for PC-G using multi-writers, and there is no solution with only single writers by theorem 5.3. Hence, multi-writers cannot be constructed from single-writers in PC-G. ■

5.2 Number of Variables

After showing that at least one multi-writer is required by a CSP solution for PC-G, a natural question is what is the minimum number of variables needed to solve CSP(n) for PC-G?

Theorem 5.5 *There does not exist an algorithm that uses fewer than n single-writers and one multi-writer and solves CSP(n) for PC-G, for any* $n \geq 2$.

Proof: Assume that there is an algorithm A that uses fewer than n single-writers and one multi-writer and solves CSP(n) for PC-G. Since there are n processes, the pigeon-hole principle ensures that there is at least one process, say p, that does not write to any single-writer variable. Computations 1 and 2 must exist. We show that Computation 3 satisfies PC-G.

Let o_i^q be q's first write to the multi-writer. The following are the required PC-G linearizations for p and q.

$$(O|p \cup O|w, \xrightarrow{L_p}) = \langle o_1^p, \cdots, o_k^p, (o_1^q, \cdots, o_l^q)|w \rangle$$

$$(O|q \cup O|w, \xrightarrow{L_q}) = \langle o_1^q, \cdots, o_{i-1}^q, (o_1^p, \cdots, o_k^p)|w, o_i^q, \cdots, o_l^q \rangle.$$

Both sequences maintain program order. Moreover, p's sequence is valid because it consists of Computation 1 followed by only writes by q. Also, q's sequence is valid because the segment $o_1^q, \cdots, o_{i-1}^q$ does not contain any writes to the multi-writer. Since p does not write to the single-writer, the segment $(o_1^p, \cdots, o_k^p)|w$ contains only writes to the multi-writer. The segment $o_i^q, \cdots, o_l^q$ starts with a write to the multi-writer over-writing any changes the segment $(o_1^p, \cdots, o_k^p)|w$ caused. Therefore both are linearizations.

Also, each linearization lists p's writes to the multi-writer followed by q's. Since only q writes to any single-writers, the two linearizations also agree on the order of this variable. So, both linearizations agree on the order of writes for each variable (Condition 2 of Definition 2.3). ∎

When $n = 2$, the bound of theorem 5.5 is tight, even if all variables are allowed to be multi-writers.

Theorem 5.6 *Two variables are insufficient to solve CSP(2) for PC-G.*

Proof: Assume that there is an algorithm A that uses exactly 2 variables, say x and y, (even multi-writers) and solves CSP(2) for PC-G. Then, computations 1 and 2 exist. We show that Computation 3 satisfies PC-G.

Partition p's computation of Computation 3 into subsequences $S_0^p, S_1^p, \ldots, S_u^p$ where each subsequence S_i^p is defined by:

1. S_0^p contains all operations from o_1^p up to but not including the first write by p, labeled $o_{\alpha_1}^p$.
2. S_i^p, $i \geq 1$, contains all operations from $o_{\alpha_i}^p$ up to but not including the first write, labeled $o_{\alpha_{i+1}}^p$, such that $o_{\alpha_i}^p$ and $o_{\alpha_{i+1}}^p$ are applied to different variables.

Partition q's computation of Computation 3 into subsequences $S_0^q, S_1^p, \ldots, S_r^q$ similarly.

The subsequence S_0^p is either empty or consists entirely of reads returning initial values. Each subsequence S_i^p ($i \geq 1$) starts with a write and all the writes in S_i^p are applied to the same variable. If the writes in S_i^p are applied to x, S_i^p is called x-gender; otherwise, it is called y-gender. Note that S_i^p (S_i^q) alternate in gender.

To show that Computation 3 satisfies PC-G, we consider two cases (the other two cases are symmetric).

S_1^p is an x-gender but S_1^q is a y-gender: Define $(O|p \cup O|w, \xrightarrow{L_p})$ and $(O|q \cup O|w, \xrightarrow{L_q})$ as follows.

$$(O|p \cup O|w, \xrightarrow{L_p}) \;=\; \langle S_0^p,\ (S_0^q)|w,\ S_1^p,\ (S_1^q)|w,\ S_2^p,\ \cdots,\ (S_i^q)|w,\ S_{i+1}^p,\ \cdots \rangle$$

$$(O|q \cup O|w, \xrightarrow{L_q}) \;=\; \langle S_0^q,\ (S_0^p)|w,\ S_1^q,\ (S_1^p)|w,\ S_2^q,\ \cdots,\ (S_i^p)|w,\ S_{i+1}^q,\ \cdots \rangle$$

Clearly, $(O|p \cup O|w, \xrightarrow{L_p})$ and $(O|q \cup O|w, \xrightarrow{L_q})$ maintain program order. They are also valid because, for each $i \geq 1$, S_i^p (respectively, S_i^q) is of the same gender as S_{i+1}^q (respectively, S_{i+1}^p). Since S_i^q and S_{i+1}^p are of the same gender, adding $(S_i^q)|w$ immediately before S_{i+1}^p does not affect p's computation because S_{i+1}^p starts with a write that obliterates the changes caused by $(S_i^q)|w$; similarly for S_i^p and S_{i+1}^q.

Algorithm	Year	$\|P\|$	Variables	`flag` Values	Fairness Delay
Dekker's	1965	$n=2$	$n+1$	2	∞
Dijkstra's	1965	$n \geq 2$	$n+1$	3	∞
Knuth's	1966	$n \geq 2$	$n+1$	3	$2^{n-1}-1$
De Bruijn's	1967	$n \geq 2$	$n+1$	3	$(n^2-n)/2$
Eisenberg and MacGuire's	1972	$n \geq 2$	$n+1$	3	$n-1$
Burns'	1981	$n \geq 2$	$n+1$	2	∞
Peterson's	1981	$n \geq 2$	$2n-1$	2	$(n^2-n)/2$

Figure 1: Well known CSP algorithms for SC

The order on the writes to x in p's linearization is:

$$(S_1^p)|w,\ (S_2^q)|w,\ \cdots,\ (S_i^p)|w,\ (S_{i+1}^q)|w,\ \cdots,\ \text{(where } i \text{ is odd)}$$

which is the same order maintained by q's linearization. The same applies to y. Therefore, Condition 2 of Definition 2.3 is also satisfied.

S_1^p and S_1^q are both x-gender: Define $(O|p \cup O|w, \xrightarrow{L_p})$ and $(O|q \cup O|w, \xrightarrow{L_q})$ as follows.

$$(O|p \cup O|w, \xrightarrow{L_p}) \ = \ \langle (S_0^q)|w,\ S_0^p,\ (S_1^q)|w,\ S_1^p,\ \cdots,\ (S_i^q)|w,\ S_i^p,\ \cdots \rangle$$

$$(O|q \cup O|w, \xrightarrow{L_q}) \ = \ \langle S_0^q,\ S_1^q,\ (S_0^p)|w,\ S_2^q,\ (S_1^p)|w,\ S_3^q,\ \cdots,\ (S_i^p)|w,\ S_{i+2}^q,\ \cdots \rangle$$

Similar analysis to the previous case shows that these are PC-G linearizations.

Thus, in all cases, Computation 3 is PC-G, and our assumption must have been in error. ∎

Since at least one multi-writer is necessary to solve CSP for PC-G, and since two multi-writers are insufficient to solve CSP(2) for PC-G, and since Peterson's Algorithm for CSP(2) uses exactly two single-writers and one multi-writer, we conclude the following.

Corollary 5.7 *Two single-writers and one multi-writer are the necessary and sufficient number and type of variables required to solve CSP(2) for PC-G.*

5.3 On the General Case

By theorems 5.3 and 5.5, an algorithm that solves CSP(n) for PC-G must use at least n single-writers and one multi-writer. Most algorithms that solve CSP(n) for SC use exactly this number and type of variables. In particular, all the algorithms discussed in this section (except Peterson's which uses n single-writers and $n-1$ multi-writers) use the same number of variables: one multi-writer (`turn`) and n single-writers. Furthermore, each process writes and reads `turn`, and each process i is associated with the single-writer `flag[`i`]`. Every process $j \neq i$ reads `flag[`i`]`. These algorithms are quoted in Appendix A and listed in Figure 1, which characterizes each algorithm by four attributes: number of processes $|P| = n$, number of variables, number of values that a `flag` variable can be assigned, and delay. The delay is an upper bound on the number of times processes enter their critical sections before a certain process gets the opportunity to enter its critical section. When the there is no upper bound on the fairness delay (∞), the algorithm is prone to starvation, and is thus unfair.

Although this number of variables is a necessary requirement for a PC-G solution, we show next that most of these algorithms do not solve CSP(n) for PC-G. First, we provide some *rules-of-thumb* that allows us to nail down certain properties of correct solutions for PC-G. Then, these rules are used to show that Dekker's, Dijkstra's, Knuth's, De Bruijn's, and Eisenberg and MacGuire's fail to solve CSP(n) for PC-G.

Lemma 5.8 *Any algorithm that uses exactly n single-writers and one multi-writer and solves CSP(n) for PC-G must satisfy each of the following properties:*

1. *Each process writes one single-writer at least once in <entry>.*
2. *Each process must write the multi-writer at least once in <entry>, and this write cannot be the last operation in <entry>.*
3. *Each process must read every other single-writer in <entry>.*

Proof: We follow the proof template given in Section 4.

1. Assume it is not the case; then there is at least one process, say p, that does not write to any single-writer. The linearizations used in Theorem 5.5 apply.
2. Assume that a process p either does not write the multi-writer in <*entry*> or does write the multi-writer exactly once and this write operation is o_k^p. Under this assumption, Computation 3 satisfies PC-G as shown by the following linearizations.

$$(O|p \cup O|w, \xrightarrow{L_p}) = \langle o_1^p, ..., o_{k-1}^p, (o_1^q, ..., o_l^q)|w, o_k^p \rangle$$

$$(O|q \cup O|w, \xrightarrow{L_q}) = \langle o_1^q, ..., o_l^q, (o_1^p, ..., o_k^p)|w \rangle$$

   Both maintain program order and are valid. They also maintain the same order on the writes to the multi-writer, which is simply q's writes then o_k^p. Note that this case is equivalent to the case where multi-writer is written in the <*critical section*> rather than in <*entry*>.

3. Assume, for the sake of contradiction, that there is a process, q, that does not read some single-writer of another process p. The linearizations of Theorem 5.5 apply.

■

Corollary 5.9 *The following CSP algorithms do not solve CSP(n) for PC-G, even if $n = 2$:*

1. *Dijkstra's Algorithm*
2. *Dekker's Algorithm*
3. *De Bruijn's Algorithm*
4. *Knuth's Algorithm*
5. *Eisenberg and MacGuire's Algorithm*

Proof: First, note that all these algorithms use n single-writers and one multi-writer.

In Dijkstra's Algorithm, if the multi-writer `turn` is initially p, p enters its <*critical section*> without writing to the multi-writer. In Dekker's and Bruijn's algorithms, the multi-writer is only written in <*exit*>. In Knuth's, and in Eisenberg and MacGuire's algorithms, the multi-writer is only written as the last step in <*entry*>. By Lemma 5.8(2), all of these algorithms are incorrect for PC-G.

■

Theorem 5.10 *Burns' Algorithm is an unfair CSP(n) solution for PC-G.*

Proof: *Mutual Exclusion:* Assume for the sake of contradiction that there exists some PC-G computation of Burns' Algorithm where two processes, say i and j, execute in their <*critical section*> concurrently. Then, i (respectively, j) must read `flag[j]` (respectively, `flag[i]`) to be *false* at line 11 before entering its <*critical section*> as shown by the following computation.

$$\textbf{Computation 4} \begin{cases} i: \ \ldots \ r(\texttt{flag[}j\texttt{]})\mathit{false} \ < \ \mathit{critical\ section} \ > \\ j: \ \ldots \ r(\texttt{flag[}i\texttt{]})\mathit{false} \ < \ \mathit{critical\ section} \ > \end{cases}$$

Note that any time a process, say i, executes a $w(\texttt{flag[}i\texttt{]})\mathit{true}$, the next operation it executes is a $w(\texttt{turn})i$. Let $w(\texttt{turn})i$ be the last write operation to `turn` that i executes before entering its <*critical section*> (This write could be performed at line 2 or 8.) Similarly, let $w(\texttt{turn})j$ be the last write to `turn` that j did before entering its <*critical section*>.

Since Computation 4 satisfies PC-G, there must exist two linearizations, $(O|i \cup O|w, \xrightarrow{L_i})$ and $(O|j \cup O|w, \xrightarrow{L_j})$, such that both agree on the order of writes to `turn`. Without loss of generality, suppose $w(\texttt{turn})i$ precedes $w(\texttt{turn})j$ in both linearizations. Since $w(\texttt{turn})j \xrightarrow{L_j} r(\texttt{flag[}i\texttt{]})\mathit{false}$ (by program order), $w(\texttt{turn})i \xrightarrow{L_j} r(\texttt{flag[}i\texttt{]})\mathit{false}$. There must be some write $w(\texttt{flag[}i\texttt{]})\mathit{true}$, such that this write is the last write by i that precedes $w(\texttt{turn})i$ in j's view. Since $w(\texttt{turn})i$ is the last write by i before it enters its <*critical section*>, $w(\texttt{flag[}i\texttt{]})\mathit{true}$ must be the last write to `flag[i]` before i enters its <*critical section*>. By transitivity, this write is the most recent write to `flag[i]` that precedes $r(\texttt{flag[}i\texttt{]})\mathit{false}$ in j's view, contradicting the validity of $(O|j \cup O|w, \xrightarrow{L_j})$. Therefore, Burns' algorithm satisfies Mutual Exclusion for PC-G.

Progress: If only one process is participating, then it will enter the <*critical section*>. So assume m processes, $2 \leq m \leq n$, are participating in a computation of Burns' Algorithm such that none of them is able to progress to <*critical section*>. We show this is impossible. By PC-G, all processes must agree of the order of the writes to `turn`, and eventually $m-1$ of them will see `turn` different from their own identifiers; therefore, all $m-1$ processes enter the body of the while loop. At least one process will fail the test on line 4 skipping the while loop. This is because of the total order on the writes to `turn` that all processes agree on. Since there is at least one process, say j, that does not engage in the while loop, we must have the following, where $i \neq j$:

$$w(\texttt{turn})i \xrightarrow{L_i} w(\texttt{turn})j \xrightarrow{L_i} r(\texttt{turn})j.$$

Since $w(\texttt{flag[}j\texttt{]})\mathit{true}$ precedes $w(\texttt{turn})j$ in program order, we conclude:

$$w(\texttt{flag[}j\texttt{]})\mathit{true} \xrightarrow{L_i} r(\texttt{flag[}j\texttt{]})\mathit{true}.$$

Therefore, lines 7 and 8 are unreachable for i unless j makes progress to <*exit*>. So, i is repeatedly executing lines 4 and 5 and $w(\texttt{flag[}i\texttt{]})\mathit{false}$ of line 5 must eventually appear in $(O|j \cup O|w, \xrightarrow{L_j})$, and consequently j enters its <*critical section*>.

```
n Processes

    shared objects
    flag[0 .. n−1] in {true, false}, single-writer
    turn in {0,···,n−1}, multi-writer

    <entry>
    flag[i] ← true
    turn ← i
    repeat
          while (turn ≠ i) do
                flag[i] ← false
                if (∀j ≠ i, not flag[j]) then
                  flag[i] ← true
                  turn ← i
                end-if
          end-while
    until (∀j ≠ i, not flag[j])

    <critical section>

    <exit>
    flag[i] ← false
```

Figure 2: Burns' CSP unfair solution

Processes have unique identifiers from the set $\{0,\cdots,n-1\}$, where n is the total number of processes. The algorithm is given by specifying the <*entry*> and <*exit*> sections of process i, $i \in \{0,\cdots,n-1\}$.

Fairness: To see that Burns' algorithm is unfair for PC-G, we show it's unfair even for SC.[4] Consider the Computation 5 which represents a starvation scenario, where the segments enclosed by square brackets can be repeated indefinitely.

$$\textbf{Computation 5} \begin{cases} i: \ [w(\texttt{flag[}i\texttt{]})true\ w(\texttt{turn})i\ r(\texttt{turn})i\ r(\texttt{flag[}j\texttt{]})false \\ \qquad < \ critical\ section\ > \ w(\texttt{flag[}i\texttt{]})false] \\ j: \ w(\texttt{flag[}j\texttt{]})true\ w(\texttt{turn})j\ [r(\texttt{turn})i\ w(\texttt{flag[}j\texttt{]})false \\ \qquad r(\texttt{flag[}i\texttt{]})true] \end{cases}$$

The following is an SC linearization. $(O, \xrightarrow{L}) = \langle w_j(\texttt{flag[}j\texttt{]})true\ w_j(\texttt{turn})j\ [w_i(\texttt{flag[}i\texttt{]})true\ w_i(\texttt{turn})i\ r_i(\texttt{turn})i\ r_j(\texttt{turn})i\ w_j(\texttt{flag[}j\texttt{]})false\ r_j(\texttt{flag[}i\texttt{]})true\ r_i(\texttt{flag[}j\texttt{]})false\ <\ critical\ section\ >\ w_i(\texttt{flag[}i\texttt{]})false]$. Operations are subscripted by the corresponding process id. The segment enclosed in square brackets is the part of the computation being repeated indefinitely.

■

6 Conclusion

Any solution to CSP(n) for PC-G must use at least one multi-writer and n single-writers. This lower bound is tight when $n = 2$. Burns' algorithm, which uses one multi-writer and n single-writers is an unfair solution

[4]It is common knowledge that Burns' algorithm is unfair even for SC.

for PC-G. It is not clear to us yet whether a fair solution can be constructed using only one multi-writer and n single-writers. If not, then to tighten the lower bound in the general case, impossibility proofs will have to exploit fairness. Many other algorithms that use the same number and type of variables as Burns' have been shown to fail for PC-G. Finally, Peterson's algorithm, which uses $n-1$ multi-writers and n single-writers, is correct and fair for PC-G.

References

[1] M. Ahamad, R. Bazzi, R. John, P. Kohli, and G. Neiger. The power of processor consistency. In *Proc. 5th Int'l Symp. on Parallel Algorithms and Architectures*, pages 251–260, June 1993. Technical Report GIT-CC-92/34, College of Computing, Georgia Institute of Technology.

[2] H. Attiya, S. Chaudhuri, R. Friedman, and J. L. Welch. Shared memory consistency conditions for non-sequential execution: Definitions and programming strategies. *SIAM Journal of Computing*, 27(1):65–89, February 1998.

[3] J. E. Burns. Symmetry in systems of asynchronous processes. In *Proc. 22nd Symp. on Foundations of Computer Science*, pages 169–174, 1981.

[4] J. Goodman. Cache consistency and sequential consistency. Technical Report 61, IEEE Scalable Coherent Interface Working Group, March 1989.

[5] G. J. Holzmann. The model checker SPIN. *IEEE Trans. on Software Engineering*, 23(5):1–5, May 1997.

[6] J. Kawash. *Limitations and Capabilities of Weak Memory Consistency Systems*. Ph.D. dissertation, Department of Computer Science, The University of Calgary, January 2000.

[7] L. Lamport. A new solution of Dijkstra's concurrent programming problem. *Communication of the ACM*, 17(8):453–455, August 1974.

[8] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. on Computers*, C-28(9):690–691, September 1979.

[9] L. Lamport. The mutual exclusion problem (parts I and II). *Journal of the ACM*, 33(2):313–326 and 327–348, April 1986.

[10] R. J. Lipton and J. S. Sandberg. PRAM: A scalable shared memory. Technical Report 180-88, Department of Computer Science, Princeton University, September 1988.

[11] G. L. Peterson. Myths about the mutual exclusion problem. *Information Processing Letters*, 12(3):115–116, 1981.

[12] M. Raynal. *Algorithms for Mutual Exclusion*. The MIT Press, 1986.

[13] A. Silberschatz, J. L. Peterson, and P. B. Galvin. *Operating System Concepts*. Addison Wesley, 1991.

[14] P. M. B. Vitanyi and B. Awerbuch. Atomic shared register access by asynchronous hardware. In *Proc. 27th Symp. on Foundations of Computer Science*, 1986.

A CSP Algorithms

For each of the following CSP algorithms, processes have unique identifiers from the set $\{0,\cdots,n-1\}$, where n is the total number of processes. The algorithms are given by specifying the <*entry*> and <*exit*> sections of process i, $i \in \{0,\cdots,n-1\}$.

A.1 Peterson's Algorithm

Two Processes

```
shared objects
flag[0 .. 1] in {true, false}, single-writer
turn in {0,1}, multi-writer

<entry>
flag[i] ← true
turn ← j
while (flag[j] and turn = j) do nothing

<critical section>

<exit>
flag[i] ← false
```

n Processes

```
shared objects
flag[0 .. n−1] in {−1 .. n−2}, single-writer
turn[0 .. n−2] in {0 .. n−1}, multi-writer

<entry>
for k = 0 to n−2 do
      flag[i] ← k
      turn[k] ← i
      while (∀j ≠ i, flag[j] ≥ k and turn[k] = i) do nothing

<critical section>

<exit>
flag[i] ← −1
```

A.2 Dekker's Two-process Algorithm

2 Processes

```
shared objects
flag[0 .. 1] in {true, false}, single-writer
turn in {0,1}, multi-writer

<entry>
flag[i] ← true
while (flag[j]) do
      if (turn = j) then
        flag[i] ← false
        while (turn = j) do nothing
        flag[i] ← true
      end-if
end-while

<critical section>

<exit>
```

```
turn ← j
flag[i] ← false
```

A.3 Dijkstra's Algorithm

n Processes

```
shared objects
flag[0 .. n−1] in {idle, requesting, in-cs}, single-writer
turn in {0,···,n−1}, multi-writer

<entry>
repeat
      flag[i] ← requesting
      while (turn ≠ i) do
            if (flag[turn] = idle) then
              turn ← i
      end-while
      flag[i] ← in-cs
until (∀ j ≠ i, flag[j] ≠ in-cs)

<critical section>

<exit>
flag[i] ← idle
```

A.4 Knuth's Algorithm

n Processes

```
shared objects
flag[0 .. n−1] in {idle, requesting, in-cs}, single-writer
turn in {0,···,n−1}, multi-writer

<entry>
repeat
      flag[i] ← requesting
      j ← turn
      while (j ≠ i) do
            if (flag[j] ≠ idle) then
              j ← turn
            else j ← (j−1) mod n
      end-while
      flag[i] ← in-cs
until (∀ j ≠ i, flag[j] ≠ in-cs)
turn ← i

<critical section>

<exit>
turn ← (i−1) mod n
flag[i] ← idle
```

A.5 De Bruijn's Algorithm

n Processes

```
shared objects
flag[0 .. n−1] in {idle, requesting, in-cs}, single-writer
turn in {0,···,n−1}, multi-writer

<entry>
repeat
      flag[i] ← requesting
```

```
        j ← turn
        while (j ≠ i) do
                if (flag[j] ≠ idle) then
                   j ← turn
                else j ← (j − 1) mod n
        end-while
        flag[i] ← in-cs
until (∀j ≠ i, flag[j] ≠ in-cs)

<critical section>

<exit>
if (flag[turn] = idle and turn = i) then
   turn ← (turn − 1) mod n
end-if
flag[i] ← idle
```

A.6 Eisenberg and MacGuire's Algorithm

```
n Processes

   shared objects
   flag[0 .. n − 1] in {idle, requesting, in-cs}, single-writer
   turn in {0, ···, n − 1}, multi-writer

   <entry>
   repeat
           flag[i] ← requesting
           j ← turn
           while (j ≠ i) do
                   if (flag[j] ≠ idle) then
                      j ← turn
                   else j ← (j + 1) mod n
           end-while
           flag[i] ← in-cs
   until ((∀j ≠ i, flag[j] ≠ in-cs) and (turn = i or flag[turn] = idle))
   turn ← i

   <critical section>

   <exit>
   j ← (turn + 1) mod n
   while (j ≠ turn and flag[j] = idle) do
           j ← (j + 1) mod n
   end-while
   turn ← j
   flag[i] ← idle
```