Critical Sections and Producer/Consumer Queues in Weak Memory Systems*†

Lisa Higham and Jalal Kawash

Department of Computer Science, The University of Calgary, Calgary, Alberta, Canada

email:<last name>@cpsc.ucalgary.ca

Abstract

In machines with weak memory consistency models, the ordering constraints on memory accesses are few. In order to properly program these machines, certain powerful explicit synchronization instructions are additionally provided by their architecture. We show that although a solution to the critical section problem (also referred to as the mutual exclusion problem) is impossible without such powerful synchronization, certain versions of the producer/consumer problem can be solved even in the weakest systems without the need for any such powerful instructions. These instructions are expensive, and avoiding their use is desirable for better performance.

1 Introduction

The problem of process coordination has been extensively addressed in the context of sequential consistency. However, modern multiprocessors present a large variety of memory models that are anything but sequentially consistent. Relaxed memory consistency models increase the challenge of solving various coordination problems. The relaxation of ordering constraints on memory accesses in these models makes reasoning about concurrency a subtle and involved issue. Process coordination problems are still more difficult if the description of the way processors interact with memory is ambiguous, informal, or incomplete.

Algorithms that coordinate processes via critical sections have long been known for sequentially consistent systems [32, 23, 28, 31]. However, many weaker memory models cannot provide this coordination through only read and write operations. Therefore, multiprocessors come equipped with additional basic operations which make synchronization possible. The use of synchronization primitives, however, incurs substantial additional execution time. So we are motivated to keep the use of strong primitives to a minimum. Hence we need to determine what kinds of coordination problems can be solved on common memory models using just read and write operations.

In earlier work we defined a formal framework for specifying memory consistency models [18]. In this paper, we use that formalism to address various process coordination problems for some memory consistency models where constraints on orderings of memory accesses are weaker than those guaranteed by sequential consistency. Specifically, this paper determines the possibility or impossibility of solving the critical section problem and versions of the producer/consumer problem on SPARC multiprocessors (with memory models Total Store Order and Partial Store Order) as well as on common weak memory models from the literature (Coherence, Pipelined-RAM, Goodman's processor consistency) without using strong synchronization operations.

Lamport's seminal work beginning in 1978 [21, 22, 24] seems to have launched investigations into relaxations of sequential consistency. Since then there has been substantial research on a large selection of memory consistency models. Most closely related to our research are several papers presenting formal descriptions of specific memory models [22, 2, 1, 4, 5, 8, 14, 13, 16, 20, 27, 30] and papers that present various formalisms for describing memories and reasoning about them [3, 6, 7, 4, 11, 15, 25, 26, 17, 29, 33]. Our work has benefited from all of these papers. For an extensive bibliography on memory consistency models see the online listing at the University of Alberta (http://www.cs.ualberta.ca/~rasit/dsmbiblio/node2.html).

Section 2 reviews a framework for defining memory consistency models, and then uses it to define six models: Sequential Consistency, Coherence, Pipelined RAM, Processor Consistency, and the SPARC Total Store Order and Partial Store Order models. Section 3 examines the ability of these models to support mutual exclusion. Section 4 examines the ability of these models to solve some producer/consumer problems. Our contributions are summarized in Section 5.

†This research was supported by a trust fund and a postgraduate scholarship from the Natural Sciences and Engineering Research Council of Canada and a research grant from The University of Calgary.

2 Modelling Multiprocessor Memories

2.1 A framework for describing memory

Our goal is to precisely capture the impact of the memory model of a multiprocessor system on the possible outcomes of computations of that system. Here we briefly overview our framework[1] for specifying this impact.

A multiprocessor system is modelled as a collection of processes operating an a collection of shared data objects. Informally, the program of each process issues a stream of invocations to a collection of abstract objects and receives a collection of responses that, from the process's point of view, are interleaved with its stream of invocations. We do not specify how the objects are implemented, how the communication proceeds or how the invocations are serviced. Instead we precisely define the constraints that the memory system imposes on the *responses observed by each process.* This is achieved by formalizing objects, processes, executions and the constraints on these executions.

One way to define a data object is to describe the object's initial state, the operations that can be applied to the object and the change of state that results from each applicable operation. As observed by Herlighy and Wing [17], it suffices to *define* a data object to be the set of all sequence of allowable operations together with their results, as follows. An *action* is a 4-tuple (op, obj, in, out) where "op" is an operation, "obj" is an object name, and "in" and "out" are sequences of parameters. The action (op, obj, in, out) means that the operation "op" with input parameters "in" is applied to the object "obj" yielding the output parameters "out". A *(data) object* is specified by a set of sequences of actions. A sequence of actions is *valid for object* x if and only if it is in the specification of x. For example, we specify a shared atomic read-write register by the set of all sequences $(o_1, o_2, ...)$ such that 1) each o_i is either a read action, denoted by a four-tuple $(R, x, \lambda, (v_i))$[2], that returns a value v_i of register x, or a write action, denoted $(W, x, (v_i), \lambda)$, that assigns a value v_i to register x, and 2) for every read action, the value returned is the same as the value written by the most recent preceding write action in the sequence.

An action (op, obj, in, out) can be decomposed into two *matching* components, namely (op, obj, in) and (op, obj, out). The component (op, obj, in) is called the *action-invocation* and (op, obj, out) is the matching *action-response.* An *event* is either an action-invocation or an action-response. Let $(e_1, e_2, ...)$ be a sequence of events. Then event e_j *follows* event e_i if and only if $i < j$ and event e_j *immediately follows* event e_i if and only if $i = j - 1$.

A *process* is a sequence of action-invocations.

A *(multiprocess) system*, (P, J), is a collection P of processes and a collection J of objects, such that for every process $p \in P$ the action-invocations of p are applied to objects in J.

A *process execution* is a (possibly infinite) sequence of events, such that each response event follows its matching invocation event. A *(multiprocess) system execution* for a system (P, J), is a collection of process executions, one for each $p \in P$.

Note that an action invocation applied to an object may not produce an explicit output. (For example, an atomic write of a register by a process may not be followed by an acknowledgment of that write and thus the output component of the write action is empty.) We assume however, that in a process execution, each invocation event is followed by a matching response event by augmenting the stream of response events with implicit acknowledgement events for each invocation that does not have an explicit response. We place these implicit responses immediately following the matching invocation, thus reflecting the semantics of the process — that the invocation is assumed to be immediately executed.

A system execution gives rise to a set of actions — namely the set of actions that result from combining each action-invocation with its matching action-response.

Let (P, J) be a multiprocess system, and O be the set of actions that result from an execution of this system. $O|p$ denotes the set of actions $o \in O$ such that invoc(o) is in the sequence of action-invocations that defines p in P. $O|x$ is the set of actions that are applied to object x in J.

We define a partial order[3] $(O, \xrightarrow{prog})$ called *program order* on the actions of a system. Let o_1 and o_2 be actions in $O|p$ for some $p \in P$. Then o_1 *program-precedes* o_2, denoted $o_1 \xrightarrow{prog} o_2$, if and only if invoc($o_2$) follows invoc($o_1$) in the definition of p. Observe that for each process $p \in P$, the program order is a total order on $O|p$.

For the definition of some memory consistency models it is necessary to distinguish the actions that change (write) a shared object from those that only inspect (read) a shared object. Let O_w be that subset of O consisting of those actions in O that update a shared object. There are also some consistency models that provide other synchronization operations. Their corresponding actions are defined when needed.

Given any set of actions O on a set of objects J, a *linearization of* O is a linear order[4] $(O, <_L)$ such that for each object $x \in J$, the subsequence $(O|x, <_L)$ of $(O, <_L)$, which contains only the actions on x, is valid for x.

In the following sections, we define various consistency conditions by stating restrictions on executions. An execution E meets some consistency condition C if the execution meets all the conditions of C. A system provides memory consistency C if every execution that can arise from the system meets the consistency condition C.

[1] the complete framework appears in [18].

[2] λ denotes the empty sequence.

[3] A partial order is a antisymmetric, transitive relation on a set. We denote a partial order by a pair (S, R) where S is a set and $R \subset S \times S$. The notation $s_1 R s_2$ means $(s_1, s_2) \in R$. When the set S is understood, R denotes the partial order. If $S' \subset S$ then (S', R) denotes the relation $(S', R \cap (S' \times S'))$.

[4] A linear order is an irreflexive partial order (S, R) such that $\forall x, y \in S$ $x \neq y$, either xRy or yRx.

2.2 Memory Model Definitions

In earlier work we defined several memory consistency models using the framework just described [18] and showed that these definitions do unambiguously capture the (sometimes less precise) informal descriptions of these models [19]. Here we repeat the definitions (without justification) of the six models investigated in this paper [22, 16, 27, 34].

Let O be the set of actions that results from the execution E of the multiprocess system (P, J).
Execution E is *sequentially consistent* if there is a linearization $(O, <_L)$ such that $(O, \stackrel{prog}{\longrightarrow}) \subseteq (O, <_L)$.
Execution E is *coherent* if for each object $x \in J$ there is some linearization $(O|x, <_{L_x})$ satisfying $(O|x, \stackrel{prog}{\longrightarrow}) \subseteq (O|x, <_{L_x})$.
Execution E is *Pipelined-RAM*, abbreviated *P-RAM*, if for each process $p \in P$ there is a linearization $(O|p \cup O_w, <_{L_p})$ satisfying $(O|p \cup O_w, \stackrel{prog}{\longrightarrow}) \subseteq (O|p \cup O_w, <_{L_p})$.
Execution E is *Processor Consistent according to Goodman*, abbreviated *PCG*, if for each process $p \in P$ there is a linearization $(O|p \cup O_w, <_{L_p})$ satisfying

1. $(O|p \cup O_w, \stackrel{prog}{\longrightarrow}) \subseteq (O|p \cup O_w, <_{L_p})$, and
2. $\forall q \in P \ (O_w \cap O|x, <_{L_p}) = (O_w \cap O|x, <_{L_q})$

In the following, $(A \uplus B)$ denotes the disjoint union of sets A and B, and if $x \in A \cap B$ then the copy of x in A is denoted x_A and the copy of x in B is denoted x_B. Let O_a denote the set of swap atomic actions and O_{sb} denote the set of store barrier actions provided by the SPARC architecture. Let O_r denote the set of actions with read semantics. Then, $O_w \cap O_r = O_a$.
Execution E satisfies *Total Store Ordering*, abbreviated *TSO*, if there exists a total order $(O_w, \stackrel{writes}{\longrightarrow})$ such that $(O_w, \stackrel{prog}{\longrightarrow}) \subseteq (O_w, \stackrel{writes}{\longrightarrow})$ and $\forall p \in P$ there exists a total order $(O|p \uplus O_w, \stackrel{merge_p}{\longrightarrow})$, satisfying:

1. $(O|p, \stackrel{prog}{\longrightarrow}) = (O|p, \stackrel{merge_p}{\longrightarrow})$, and
2. $(O_w, \stackrel{writes}{\longrightarrow}) = (O_w, \stackrel{merge_p}{\longrightarrow})$, and
3. if $w \in (O|p \cap O_w)$ (i.e. necessarily, a write by p) then $w_{O|p} \stackrel{merge_p}{\longrightarrow} w_{O_w}$, and
4. $((O|p \uplus O_w) \backslash (O_{invisible_p} \cup O_{memwrites_p}), \stackrel{merge_p}{\longrightarrow})$ is a linearization, where
   $O_{invisible_p} = \{w \mid w$ is a write by $q \neq p$ to $x \wedge \exists w' \in O|x \cap O|p \cap O_w \wedge w'_{O|p} \stackrel{merge_p}{\longrightarrow} w \stackrel{merge_p}{\longrightarrow} w'_{O_w}\}$
   $O_{memwrites_p} = \{w_{O_w} \mid w \in O|p \cap O_w\}$, and
5. if $w \in (O|p \cap O_w)$ and $a \in (O|p \cap O_a)$ and $w \stackrel{prog}{\longrightarrow} a$, then $w_{O_w} \stackrel{merge_p}{\longrightarrow} a$.

Execution E satisfies *Partial Store Ordering*, abbreviated *PSO*, if there exists a total order $(O_w, \stackrel{writes}{\longrightarrow})$ such that $\forall x$, $(O_w \cap O|x, \stackrel{prog}{\longrightarrow}) \subseteq (O_w \cap O|x, \stackrel{writes}{\longrightarrow})$ and $\forall p \in P$ there is a total order $(O|p \uplus O_w, \stackrel{merge_p}{\longrightarrow})$, satisfying items 1 through 4 of TSO and (5) if $sb \in (O|p \cap O_{sb})$ and $w, u \in (O|p \cap O_w)$ and $w \stackrel{prog}{\longrightarrow} sb$ and $sb \stackrel{prog}{\longrightarrow} u$, then $w_{O_w} \stackrel{merge_p}{\longrightarrow} u_{O_w}$.

3 Critical Section Problem

In the critical section problem (CSP), a set of processes coordinate to share a resource. We investigate minimum requirements for memory models to be able to provide a solution to CSP without the use of strong synchronization primitives.

We assume each process has the following structure:

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

Given the multiprocess system (P, J), a solution to the critical section problem must satisfy the following properties:

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

Now, we turn to some impossibilities concerning CSP. First, we need the following lemma.

Lemma 3.1 *A P-RAM system (P, J) with only single-writer objects is PCG.*

Proof: Let $<_p$ be a linearization for $p \in P$ of $(O|p \cup O_w)$ that is guaranteed by P-RAM. Similarly, let $<_q$ be a linearization for $q \in P$ over $(O|q \cup O_w)$. Since, for any object $x \in J$, there is only one processor say r that writes to x, and both $<_p$ and $<_q$ have all these writes to x in the program order of r, the order of the writes to x in $<_p$ is the same as the order of the writes to x in $<_q$. Therefore, the definition of PCG is satisfied. ∎

We will use partial executions Ex1, Ex2, and Ex3 defined as follows.

$$\textbf{Ex1} \ \begin{cases} p : o_1^p, o_2^p, ..., o_k^p & (p \textit{ in } < \textit{critical section} >) \\ q : \lambda \end{cases}$$

where λ denotes the empty sequence and o_i^p denotes the i^{th} operation of p.

$$\textbf{Ex2} \ \begin{cases} p : \lambda \\ q : o_1^q, o_2^q, ..., o_l^q & (q \textit{ in } < \textit{critical section} >) \end{cases}$$

$$\textbf{Ex3} \ \begin{cases} p : o_1^p, o_2^p, ..., o_k^p & (p \textit{ in } < \textit{critical section} >) \\ q : o_1^q, o_2^q, ..., o_l^q & (q \textit{ in } < \textit{critical section} >) \end{cases}$$

Claim 3.2 *There is no algorithm even for two processes that solves CSP in any of the following systems:*

1. *a PCG system with only single-writer objects.*
2. *a system that is only P-RAM and Coherent but not PCG*[5].
3. *a TSO system without the use of swap-atomic instructions, or a PSO system without the use of store-barrier instructions.*
4. *a sequentially consistent system with one shared object.*

Note: Claim 3.2(4) was previously proved by Burns and Lynch [9].

Proof: For any of these systems $(\{p, q\}, J)$, assume for the sake of contradiction that there is a mutual exclusion algorithm A that solves CSP. If A runs with p in $< entry >$ and with q staying in $< remainder >$ then the Progress property ensures that we must get an execution of the form of Ex1. Similarly, if A runs with q in $< entry >$ and without p participating, we must get an execution of the form of Ex2. We then use the existence of executions Ex1 and Ex2 to build an execution of the form of Ex3 and show that it satisfies the definition of each one of these systems. Hence Ex3 is a possible execution of A. However, in Ex3 both p and q are in their critical sections simultaneously, contradicting the requirement that A satisfies Mutual Exclusion.

1. Let $(o_1^p, ..., o_k^p)((o_1^q, ..., o_l^q)|_w)$ be a total order of $(O|p \cup O_w)$ and let $(o_1^q, ..., o_l^q)((o_1^p, ..., o_k^p)|_w)$ be a total order of $(O|q \cup O_w)$. Clearly each preserves $\xrightarrow{prog}$. Also, each is a linearization because the first part corresponds to a possible execution, and the second part contains only writes. Therefore, Ex3 is P-RAM. However, by Lemma 3.1, Ex3 is also PCG if it uses only single-writer objects.
2. As shown in item 1, Ex3 is P-RAM. To prove that Ex3 is Coherent, let o_j^p be the first write to object x by p. Then, $(o_1^p, ..., o_{j-1}^p)|_x \ (o_1^q, ..., o_l^q)|_x \ (o_j^p, ..., o_k^p)|_x$ is a total order on $O|x$ that clearly preserves $\xrightarrow{prog}$. Each of the segments $(o_1^p, ..., o_{j-1}^p)|_x$, $(o_j^p, ..., o_k^p)|_x$ and $(o_1^q, ..., o_l^q)|_x$ occur in a possible execution; the first two are concatenated before there has been any write to x and the first operation of the last segment is a write to x which obliterate any previous change to x by q. Therefore the sequence is a linearization of $O|x$, showing that Ex3 is Coherent.
3. To see that Ex3 satisfies TSO, we imagine a situation where p and q enter their critical sections before the contents of their write-buffers are committed to main memory. Specifically, let $(O_w, \xrightarrow{writes})$ be $(o_1^p, ..., o_k^p, o_1^q, ..., o_l^q)|_w$ which clearly preserves $\xrightarrow{prog}$. Let $\xrightarrow{merge_p}$ be $o_1^p, ..., o_k^p$, $(o_1^p, ..., o_k^p,\ o_1^q, ..., o_l^q)|_w$, and $\xrightarrow{merge_q}$ be $o_1^q, ..., o_l^q$, $(o_1^p, ..., o_k^p,\ o_1^q, ..., o_l^q)|_w$, Conditions 1, 2, 3, and 5 of TSO are obviously satisfied. Condition 4, follows because, for p, $((O|p \uplus O_w) \setminus (O_{invisible_p} \cup O_{memwrites_p}) = o_1^p, ..., o_k^p$ which is clearly a linearization since it is Ex1, and similarly for q.
4. Let x be the one shared object and let o_j^p be the first write to x by p, if any. Then the sequence $(o_k^p, ..., o_{j-1}^p)(o_1^q, ..., o_l^q)(o_j^p, ..., o_j^p, ..., o_k^p)$ is a total order on O which preserves $\xrightarrow{prog}$. By a similar argument to that in item 2, this sequence, when restricted to operations on x, is a linearization. However, there are no other shared objects, hence the whole sequence is a linearization. Thus Ex3 is Sequentially Consistent if there is only one shared object.

∎

Observe that none of the arguments in Claim 3.2 depends on the Fairness property, so the systems listed there cannot solve CSP even without fairness. A second observation follows after recalling that Peterson's Algorithm [31], which uses multi-writer registers, is a correct solution for the CSP even for PCG [4]. However, Claim 3.2(1) establishes that CSP is impossible in PCG with single-writer objects.

Corollary 3.3 *In a PCG system, multi-writer objects cannot be implemented from single-writer objects.*

4 Producer/Consumer Problem

Producer/Consumer [10] objects are frequently used for process coordination. The producer is a process that produces items and places them in a shared structure. A consumer is a process which removes these items from the structure. We distinguish two structures whose solution requirements vary: the set structure where the order of consumption is insignificant, and the queue structure where items are consumed in the same order as that in which they were produced.

Producers and consumers are assumed to have the following form:

repeat
    <entry>
    <producing> (or <consuming>)
    <exit>
until *false*

We denote the producer/consumer queue problem as P_mC_n-queue where m and n are respectively the number of producer and consumer processes. Similarly the producer/consumer set problem is denoted P_mC_n-set.

A solution to P_mC_n-set must satisfy the following:

[5] A system that is both P-RAM and Coherent is not necessarily PCG[18].

- **Safety:** Every produced item is eventually consumed exactly once, and every consumed item is a produced item.

- **Progress:** Any producer in <*entry*> will eventually be in <*producing*> and then subsequently in <*exit*>. Any consumer in <*entry*> will eventually be in <*consuming*> then subsequently in <*exit*> unless the set is empty.

A solution to P_mC_n-queue must satisfy Safety and Progress plus the following property:

- **Ordered Consumption:** consumers consume items in the same order as that in which the items were produced.

Claim 4.1 *There is an algorithm that solves P_1C_1-queue in the following systems:*

1. *a system that is only Coherent.*
2. *a system that is only P-RAM.*

Proof: We show that the algorithm in Figure 1 is correct.

1. A Coherent system:
   Safety : line (1) is the producer's only write to global memory. By coherence, this global write can not be committed to main memory unless "queue[in] = $\perp$" since these two operations are to the same object. Therefore, program order is preserved. By a similar argument, we can show that the consumer can not consume an unproduced item.

   progress: the writes in lines (1) and (3) will eventually reach the global memory and this will prevent lock-outs.

   Ordered Consumption: By coherence, two instances of line (1) may reach the global memory out of order only if they are to two different locations (different queue cells). However, the consumer will not consume items out of order because it is only required to consume the cell pointed to by "out".

2. A P-RAM system:
   Safety: the write of line (1) can not be committed to global memory unless "queue[in] = $\perp$" because P-RAM requires $\xrightarrow{prog}$ to be maintained per process. By a similar argument, the consumer can not consume an unproduced item.

   progress: as in coherence.

   Ordered Consumption: The order of consumption follows that of production because P-RAM requires the consumer to see all the writes in the system in an order which adheres to $\xrightarrow{prog}$.

■

Although CSP cannot be solved in several weak memory systems (Claim 3.2), P_1C_1-queue can be solved in many of these systems.

Corollary 4.2 *There is an algorithm that solves P_1C_1-queue in a PCG system with only single-writer objects.*

Proof: Since PCG systems are both coherent and P-RAM, the corollary follows from Claim 4.1. ■

Corollary 4.3 *There is an algorithm that solves P_1C_1-queue in a TSO system even without the use of swap-atomic instructions, or a PSO system even without the use of store-barrier instructions.*

Proof: TSO and PSO systems are coherent [19]. Therefore the corollary follows from Claim 4.1. ■

We have now seen weak memory systems for which CSP cannot be solved, but P_1C_1-queue (and hence P_1C_1-set) can be. We now investigate more general producer-consumer problems for weak systems.

Claim 4.4 *There is no solution for P_1C_n-queue or P_mC_1-queue ,even for $n = 2$ or $m = 2$, in any of the following systems:*

1. *a PCG system with only single-writer objects.*
2. *a system that is only P-RAM and Coherent but not PCG.*
3. *a TSO system without the use of swap-atomic instructions, or a PSO system without the use of store-barrier instructions.*
4. *a sequentially consistent system with one shared object.*

Proof: The proof is similar to that of Claim 3.2.
P_1C_n-queue:
Suppose there is a solution to P_1C_n-queue for some system, with producer p and two consumers c and d. Consider Ex4 where p places item ι in the queue and quits, then c removes item ι while d is idle:

$$\textbf{Ex4}\ \begin{cases} p : o_1^p, o_2^p, ..., o_k^p & (\textit{p has produced item } \iota.) \\ c : o_1^c, o_2^c, ..., o_l^c & (\textit{c has consumed item } \iota.) \\ d : \lambda \end{cases}$$

By Progress, this execution must be possible in the system. Similarly the following execution is also possible in the system:

$$\textbf{Ex5}\ \begin{cases} p : o_1^p, o_2^p, ..., o_k^p & (\textit{p has produced item } \iota.) \\ c : \lambda \\ d : o_1^d, o_2^d, ..., o_j^d & (\textit{d has consumed item } \iota.) \end{cases}$$

Notice that the sequence for p is identical in both Ex4 and Ex5, and that p completes this sequence before c or d begin.

For each of the systems listed above, we will show that if Ex4 and 5 are possible then so is the execution:

$$\textbf{Ex6}\ \begin{cases} p : o_1^p, o_2^p, ..., o_k^p & (\textit{p has produced item } \iota.) \\ c : o_1^c, o_2^c, ..., o_l^c & (\textit{c has consumed item } \iota.) \\ d : o_1^d, o_2^d, ..., o_j^d & (\textit{d has consumed item } \iota.) \end{cases}$$

```
shared var: queue: array[0..n − 1] of item (initialized to ⊥)

— the producer                              — the consumer
    var                                         var
        in : 0..n − 1                               out : 0..n − 1
        ip : item                                   ic : item

    in ← 0                                      out ← 0
    repeat                                      repeat

    < entry >                                   < entry >
          if queue[in] = ⊥ then                       if queue[out] ≠ ⊥ then
    < producing >                               < consuming >
              ... produce ip                              ic ← queue[out]    (2)
              queue[in] ← ip    (1)             < exit >
    < exit >                                              queue[out] ← ⊥    (3)
              in ← in + 1 mod n                           out ← out + 1 mod n
          fi                                          fi
    until false                                 until false
```

Figure 1: P_1C_1*-queue Algorithm*

However, in Ex6 both c and d have consumed the same item, contradicting the Safety requirement. Thus we can conclude that a solution to P_1C_n-queue is impossible in that system.

1. Let $o_1^p, ..., o_k^p$, $(o_1^c, ..., o_l^c, o_1^d, ..., o_l^d)|_w$ be a total order of $(O|p \cup O_w)$ and let $(o_1^p, ..., o_k^p)|_w$, $o_1^c, ..., o_l^c$, $(o_1^d, ..., o_l^d)|_w$ be a total order of $(O|c \cup O_w)$, and let $(o_1^p, ..., o_k^p)|_w$, $o_1^d, ..., o_l^d$, $(o_1^c, ..., o_l^c)|_w$ be a total order of $(O|d \cup O_w)$. Clearly each preserves $\xrightarrow{prog}$. Also, each is a linearization because each processor behaves exactly as it did in Ex4 (or Ex5) where it acted alone and is followed by a segment containing only writes. Therefore, Ex6 is P-RAM. However, by Lemma 3.1, Ex6 is also PCG if it uses only single-writer objects.

2. As shown in item 1, Ex6 is P-RAM. To prove that Ex6 is Coherent, let o_m^c be the first write to object x by c. Then, $(o_1^p, ..., o_k^p, o_1^c, ..., o_{m-1}^c, o_1^d, ..., o_j^d, o_m^c, ..., o_l^c)|_x$ is a total order on $O|x$ that clearly preserves $\xrightarrow{prog}$. Each of the seqments $(o_1^p, ..., o_k^p, o_1^c, ..., o_{m-1}^c)|_x$, and $(o_m^c, ..., o_l^c)|_x$ and $(o_1^d, ..., o_j^d)|_x$ occur in a possible execution; the first two are concatenated before there has been any write to x by c so that the computation of d on x is unchanged from Ex5. The last segment begins with a write to x which obliterate any previous change to x by d so that the computation of c on x is unchanged from Ex4. Therefore the sequence is a linearization of $O|x$, showing that Ex6 is Coherent.

3. To see that Ex6 satisfies TSO, we imagine a situation where p completes its computation and its write-buffer is emptied first, then both c and d consume item ι before the contents of their write-buffers are committed to main memory. Specifically, let $(O_w, \xrightarrow{writes})$ be $(o_1^p, ..., o_k^p, o_1^c, ..., o_l^c, o_1^d, ..., o_j^d)|_w$ which clearly preserves $\xrightarrow{prog}$. Let $\xrightarrow{merge_p}$ be $o_1^p, ..., o_k^p$, $(o_1^p, ..., o_k^p, o_1^c, ..., o_l^c, o_1^d, ..., o_j^d)|_w$, and $\xrightarrow{merge_c}$ be $(o_1^p, ..., o_k^p)|_w$, $o_1^c, ..., o_l^c$, $(o_1^c, ..., o_l^c, o_1^d, ..., o_j^d)|_w$, and $\xrightarrow{merge_d}$ be $(o_1^p, ..., o_k^p)|_w, o_1^d, ..., o_j^d, (o_1^c, ..., o_l^c, o_1^d, ..., o_j^d)|_w$. Conditions 1, 2, 3, and 5 of TSO are obviously satisfied. Condition 4 follows by an argument similar to that in Claim 3.2 (3).

4. Let x be the one shared object and let o_m^c be the first write to x by c, if any. Then the sequence $(o_k^p, ..., o_k^p, o_1^c, ..., o_{m-1}^c, o_1^d, ..., o_j^d, o_m^c, ..., o_l^c)$ is a total order on O which preserves $\xrightarrow{prog}$. By a similar argument to that in item 2, this sequence, when restricted to operations on x, is a linearization. However, there are no other shared objects, hence the whole sequence is a linearization. Thus Ex6 is Sequentially Consistent if there is only one shared object.

P_mC_1-queue:
The proof for the impossibility of this problem is very similar to that for P_1C_n-queue. ■

Note that any system that can solve mutual exclusion can solve P_mC_n-queue as well. This can be achieved by simply protecting the queue structure in a critical section so that it is accessed by only one process at a time. Since a PCG system can solve CSP, Claim 4.4(2) is as strong as possible.

Table 1: *Summary of Possibilities/Impossibilities*

	CSP	P_1C_1-queue	P_1C_n-queue	P_mC_1-queue	P_mC_n-queue	P_mC_n-set
TSO or PSO with basic objects	no	yes	no	no	no	yes
P-RAM & Coherent but not PCG	no	yes	no	no	no	yes
PCG with only single-writers	no	yes	no	no	no	yes
PCG with multi-writers [4]	yes	yes	yes	yes	yes	yes
sequential consistency	yes	yes	yes	yes	yes	yes
sequential consistency with only one shared variable	no [9]	yes	no	no	no	no

Corollary 4.5 *There is no algorithm that solves P_mC_n-queue in any of the systems of Claim 4.4 for $m + n \geq 3$.*

This corollary contrasts strongly with the producer/consumer *set* problem.

Claim 4.6 *There is an algorithm that solves P_mC_n-set in a system that is only Coherent or P-RAM.*

Proof: (Sketch) Since we can solve P_1C_1-queue in these systems, P_1C_n-set can be solved by associating a separate queue with each consumer. The producer inserts its items into these queues using any discipline, say round-robin. The correctness argument is straightforward. Similarly, P_mC_1-set can be solved by associating queues to producers. To solve the general P_mC_n-set we combine these two solutions. If $m \geq n$ let consumer c_i consume from producers p_{ik} for $ik \leq n$, and similarly for $n \geq m$. ∎

Corollary 4.7 *There is an algorithm that solves P_mC_n-set for each of the systems listed in Claim 4.4.*

Proof: Each such system is either Coherent or P-RAM. So, Claim 4.6 applies. ∎

5 Conclusion

Sequential consistency is an easy and intuitive memory consistency model to adopt. In fact, sequential consistency has usually been assumed for work in analysis of distributed algorithms. However, current and proposed multiprocessor machines implement weaker models than sequential consistency in order to enhance performance. In this paper, we have revisited the critical section problem and the producer/consumer problem in the context of weak memory systems — systems that are not sequentially consistent.

Table 1 summarizes our impossibility and possibility results. We have shown that several weak memory systems cannot support a solution to the critical section problem without additional synchronization or powerful instructions. These systems include coherence, P-RAM, TSO, and PSO systems. In spite of this, we have shown that certain versions of the producer/consumer problem can be solved without any additional synchronization. In particular, even the weakest memory models (coherence and P-RAM) can support a solution to the producer/consumer queue problem for two processes — a single producer and a single consumer. Moreover, they can support a solution for any number of processes if we remove the requirement of processing the queue in order.

Weak system architectures provide additional powerful synchronization instructions. These instructions are expensive; it is useful to identify ways to avoid using these primitives and incurring the corresponding performance degradation.

Ackowledgments

We thank the anonymous referees for their comments and we thank Brian W. Unger for suggesting the producer/consumer problem.

References

[1] S. V. Adve and M. D. Hill. Implementing sequential consistency in cache-based systems. *1990 Int'l Conf. on Parallel Processing*, pages I47–I50, August 1990.

[2] S. V. Adve and M. D. Hill. Weak ordering - a new definition. In *Proc. of the 17th Annual Int'l Symp. on Computer Architecture*, pages 2–14, May 1990.

[3] S. V. Adve and M. D. Hill. A unified formalization of four shared-memory models. *IEEE Trans. on Parallel and Distributed Systems*, 4(6):613–624, 1993.

[4] M. Ahamad, R. Bazzi, R. John, P. Kohli, and G. Neiger. The power of processor consistency. In *Proc. 5th ACM Symp. on Parallel Algorithms and Architectures*, pages 251–260, June 1993. Also available as College of Computing, Georgia Institute of Technology technical report GIT-CC-92/34.

[5] M. Ahamad, G. Neiger, J. Burns, P. Kohli, and P. Hutto. Causal memory: Definitions, implementations, and programming. *Distributed Computing*, 9:37–49, 1995.

[6] H. Attiya, S. Chaudhuri, R. Friedman, and J. Welch. Shared memory consistency conditions for non-sequential execution: Definitions and programming strategies. In *Proc. 5th ACM Symp. on Parallel Algorithms and Architectures*, pages 241–250, 1993.

[7] H. Attiya and R. Friedman. A correctness condition for high performance multiprocessors. In *Proc. 24th Annual ACM Symp. on Theory of Computing*, pages 679–690, 1992.

[8] H. Attiya and J. L. Welch. Sequential consistency versus linearizability. In *Proc. 3rd ACM Symp. on Parallel Algorithms and Architectures*, pages 304–315, 1991.

[9] J. E. Burns and N. A. Lynch. Bounds on shared memory for mutual exclusion. *Information and Computing*, 107(2):171–184, December 1993.

[10] E. W. Dijkstra. Cooperating sequential processes. Technical Report EWD-123, Technological University, Eidhoven, the Netherlands, 1965. Reprinted in [12].

[11] M. Dubois, C. Scheurich, and F. Briggs. Synchronization, coherence, and event ordering in multiprocessors. *Computer*, pages 9–21, January 1988.

[12] F. Genuys, editor. *Programming Languages.* Academic Press, 1968.

[13] K. Gharachorloo, S. Adve, A. Gupta, J. Hennessy, and M. Hill. Programming for different memory consistency models. *Journal of Parallel and Distributed Computing*, 15(4):399–407, August 1992.

[14] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proc. of the 17th Int'l Symp. on Computer Architecture*, pages 15–26, May 1990.

[15] P. B. Gibbons and M. Merritt. Specifying nonblocking shared memories. In *Proc. 4th ACM Symp. on Parallel Algorithms and Architectures*, pages 306–315, 1992.

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

[17] M. Herlihy and J. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. on Programming Languages and Systems*, 12(3):463–492, July 1990.

[18] L. Higham, J. Kawash, and N. Verwaal. Defining and comparing memory consistency models. In *Proc. of the 10th Int'l Conf. on Parallel and Distributed Computing Systems*, pages 349–356, October 1997.

[19] J. Kawash and L. Higham. Memory consistency models of bus-based multiprocessors. Technical Report 96/594/14, Department of Computer Science, The University of Calgary, October 1996.

[20] P. Kohli, G. Neiger, and M. Ahamad. A characterization of scalable shared memories. In *Proc. of the 1993 Int'l Conf. on Parallel Processing*, August 1993.

[21] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communication of the ACM*, 21(7):558–565, July 1978.

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

[23] L. Lamport. The mutual exclusion problem. *Journal of the ACM*, 33(2):327–348, April 1986.

[24] L. Lamport. On interprocess communication (parts i and ii). *Distributed Computing*, 1(2):77–85 and 86–101, 1986.

[25] L. Lamport. How to make a correct multiprocess program execute correctly on a multiprocessor. Technical Report 96, DEC, February 1993.

[26] D. Linder and J. Harden. Access graphs: A model for investigating memory consistency. *IEEE Trans. on Parallel and Distributed Systems*, 5(1):39–52, 1994.

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

[28] N. A. Lynch. *Distributed Algorithms.* Morgan Kaufmann Publishers, 1996.

[29] J. Misra. Axioms for memory access in asynchronous hardware systems. *ACM Trans. on Programming Languages and Systems*, 8(1):142–153, 1986.

[30] D. Mosberger. Memory consistency models. *ACM Operating Systems Review*, 27(1):18–26, January 1993.

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

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

[33] P. Sindhu, J.-M. Frailong, and M. Cekleov. Formal specification of memory models. Technical Report CSL-91-11, XEROX Corporation Palo Alto Research Center, December 1991.

[34] D. L. Weaver and T. Germond, editors. *The SPARC Architecture Manual version 9.* Printice-Hall, 1994.