Decentralized Semaphore Support in a Virtual Shared Memory System

Mahendra Ramachandran and Mukesh Singhal

Department of Computer and Information Science
The Ohio State University, Columbus, Ohio 43210-1277
email: {ramach,singhal}@cis.ohio-state.edu

Abstract

Distributed Shared Memory has increasingly become a desirable programming model on which to program multicomputer systems. Such systems strike a balance between the performance attainable in distributed memory multiprocessors and the ease of programming of shared memory systems. In shared memory systems, concurrent tasks communicate through shared variables and synchronization of access to shared data is an important issue. Semaphores have been traditionally used to provide this synchronization. In this paper we propose a decentralized scheme to support semaphores in a virtual shared memory system. Our method of grouping semaphores into semaphore-pages and caching a semaphore at a processor on demand eliminates the reliability and bottleneck problems associated with centralized schemes.

Key phrases: Operating Systems, Process Synchronization, Semaphores, Distributed Memory Architectures, Distributed Shared Memory.

1 Introduction

Distributed memory architectures such as hypercubes and meshes are appealing because they scale better than shared memory multiprocessors, however, they are not as easy to program as the latter. This has motivated research into what is known as Distributed

Shared Memory systems (DSM) [1, 2]. Such systems provide a shared memory programming paradigm on a distributed memory multiprocessor system or a network of computers. The rational is that the combination of the shared memory paradigm and the distributed memory architectures eases the tasks of programming, portability, while maintaining scalability.

In DSM systems, shared data or pages are cached onto processors on demand similar to the manner in which pages are brought into main memory in traditional virtual memory systems. However, the pages in DSM systems are resources that are competed for by concurrent processes that are executing on different processors. In order to improve the efficiency of program execution, it is desirable to replicate the shared pages and place copies on the various processors of the system. While this replication allows concurrent local access to the pages by the processes, the copies become outdated when one of the pages is modified by its processor. A similar problem arises in shared memory multiprocessors when multiple copies of shared data are cached by individual processors. While hardware based solutions are used in these systems to maintain the coherence of the caches, DSM systems solve this problem in software. Maintaining the coherence of the shared pages has been the focus of the research is DSM systems [2, 3, 4, 5, 6].

Aside from the issue of maintaining the coherency of shared data, it is important for a DSM system to provide mechanisms through which access to shared data by processes can be synchronized. A typical application on DSM systems consists of several concurrent processes that access shared variables for synchronization and communication and this concurrent access to the shared data must be synchronized to maintain the integrity of the shared data. In shared memory systems, these synchronization mechanisms aare based on atomic access to shared variables (eg. semaphores and monitors). In DSM systems, if the variables used for synchronization were placed in the shared pages, significant performance degradation would result due to the thrashing of the shared page between the processors as a result of false sharing. The alternative is to place them in memory that is not part of the shared page. This will require all synchronization operations to be sent to the processor in whose memory the synchronization variable has been mapped. This results in a centralized scheme.

While DSM systems have proposed schemes for synchronized access to data, these have been to maintain consistency of the shared data (i.e., lock and unlock mutual exclusion). Concurrent tasks still need a mechanism through which to synchronize or hand-shake at certain points during their execution. Thus it is still essential to provide a complete set of synchronization primitives.

We present a decentralized semaphore mechanism which eliminates the single point of failure and the bottleneck problems inherent in centralized schemes. In our scheme we group the semaphores into pages called semaphore-pages. These pages are similar to the data pages in DSM systems, however, the semaphore-pages are replicated on the processors of the system and we propose specific algorithms to maintain their consistency. Our approach enables processes to exploit the locality of reference which exists in accessing semaphore variables by caching them on their processors.

The rest of the paper is organized as follows: In the next section, we describe the existing synchronization mechanisms for DSM systems. In Section 3 we present a decentralized scheme to maintain semaphores in DSM systems, prove its correctness, and discuss its performance. We conclude the paper with some thoughts on future directions in Section 4.

2 Previous Work

The type of synchronization provided by DSM systems can be classified as software or hardware based. The software based mechanisms can be further divided into centralized and distributed schemes. In this section, we describe a few notable systems which fall into the above classes.

In the IVY system, eventcounts are used for synchronization. The choice was motivated by the fact that the Aegis operating system on which IVY was implemented already provided eventcounts. There are two ways of implementing eventcounts in IVY; using the shared virtual memory space or using remote procedure calls. In the first case, the eventcount variable's structure is allocated on a shared page of the DSM. The operations on the eventcount are done atomically by ensuring that the page will not be migrated to another processor prior to the completion of the operation. An alternate scheme is to allocate eventcounts to one or more server sites. An operation on an eventcount is then sent as a remote procedure call to the server which holds that eventcount.

In Clouds the DSM controller provides remote **P** and **V** operations on semaphores [7]. The semaphores are placed in semaphore segments along with lists of blocked processes for each of them. The **P** and **V** operations are performed on the processor where the segment is located in a centralized manner. An alternative approach is to move the entire segment of semaphores from the owner processor to another site. While this may allow for local access to some semaphores, it still is a centralized scheme.

Locks and barriers are provided in Munin [8]. To implement a distributed lock, Munin defines the processor holding the lock as the owner of the lock. When a lock is sought, the request is sent to the probable owner of the lock (which in turn might have to froward the request until it reaches the current owner of the lock). If the lock is free, it is sent to the requester. If not, then the request is queued up. If a processor is already queued for the lock, the new request is forwarded to that processor to be queued by it. Barriers are implemented in a centralized manner, however, the authors suggest that other schemes be adopted for reasons of scalabilty.

The Plus system is a hardware implementation of a DSM [9]. In this system, a "deferred synchronization" is performed. This is somewhat similar to the delayed branch used in RISC architectures in that a **P** operation is performed followed by computation that does not depend on the result of the **P** operation. Prior to executing the critical section code, the **P** operation's result is checked to ensure valid access.

Shiva, a DSM developed on an Intel iPSC/2 hypercube provides binary semaphores[10]. These semaphores are accessed by using send/receive primitives to the central semaphore controller. An alternative is to localize the semaphore on the processor that performs a **P** operation so as to exploit the locality of reference exhibited by the semaphore. Every processor in the system maintains a pointer to the current processor which might hold the semaphore. When a processor fails to access a semaphore locally, it sends a request to the processor specified in its pointer field, this request might need to be forwarded until it arrives at the processor holding the semaphore. The semaphore is then sent to the processor which requested it.

3 The Semaphore-page Approach

In our method, the semaphores in the system are grouped into pages, called semaphore-pages. A semaphore-page is divided into semaphore structures. Each structure contains either the semaphore or information regarding the location of the semaphore in the network (see Figure 1). While the semaphore-pages are replicated on each processor, the semaphores are not. By grouping the semaphores into the semaphore-pages, the processors in the system are able to update their view of the location of the semaphores when they communicate with each other.

When a process tries to access a semaphore, it first checks the semaphore-page on its processor. If the semaphore is found, the desired (**P** or **V**) operation is performed. If the semaphore is not available locally, its semaphore structure will point to the processor on which the semaphore likely to reside. A remote access is performed by sending a request to obtain the semaphore. When a processor which currently holds a semaphore receives a request, depending on the state of the semaphore it either queues the request or sends the semaphore to the requesting processor. If a processor which does not hold the semaphore receives the request, it forwards it to the processor likely to be holding the semaphore.

When processors exchange messages with each other to access a semaphore, they exchange additional information regarding the locations of all the semaphores of a particular semaphore-page. We make use of timestamps in determining which processor has the more recent information regarding the location of a particular semaphore. This enables the processors to update their semaphore-pages to reflect the current state of the system. We describe this updating of semaphore-pages in detail in Section 3.4. A description of the model and a detailed description of the approach follows.

3.1 System Model and Definitions

The system consists of n processors labeled $P_1,...,P_n$. The processors are connected by an interconnection network or a local area network and communicate via message passing primitives. It is assumed that the communications network is reliable; all messages are delivered within a finite time delay. Failure of a processor would require the system to be re-started.

There are two types of semaphores in the system; binary and resource counting. A binary semaphore is initialized to the value 1 and a resource counting semaphore is initialized to some value, N. A **P** and its corresponding **V** operation on a semaphore can be performed on two different processors.The **P** and **V** operations are defined as:

```
P(Sem):      if (Sem >=  1)
               then   Sem := Sem - 1;
               else   block the process on the semaphore queue;

V(Sem):      if (semaphore queue is not empty)
                then  de-queue a process;
                else  Sem := Sem + 1;
```

Each processor maintains a copy of p semaphore-pages, denoted as $SP_1,...,SP_p$. This however does not imply that the semaphores themselves are replicated; only one processor holds a particular semaphore. Initially, the semaphores are mapped onto different processors in some manner that is not dependent on the mechanism. The other processors maintain information pertaining to the location of that semaphore. There are m semaphores per semaphore-page, denoted by $S_1,...,S_m$. Semaphore S_j in page SP_i is identified by a tuple, $<SP_i,S_j>$, that identifies the page and the semaphore index within that page. Each semaphore in a semaphore-page consists of the following structure:

```
Sem: semaphore;
probable_holder : integer;
blocked_req  : queue of processor_id;
timestamp : integer;
requested : boolean;
```

When a semaphore is cached on a processor, its semaphore structure holds the semaphore. Otherwise, the semaphore field is NULL. The probable_holder field in the structure holds the ID of the processor where the semaphore is likely to be located. The blocked_req queue holds blocked requests which are waiting to be granted. The timestamp field associates an age with the information in the probable_holder field. Timestamps are discussed in conjunction with updating semaphore-pages in Section 3.4. The last field, requested, is true when there is an outstanding request for the semaphore and is false otherwise. Since multiple processes can execute on a processor, this field can be used block all except one request for the same semaphore.

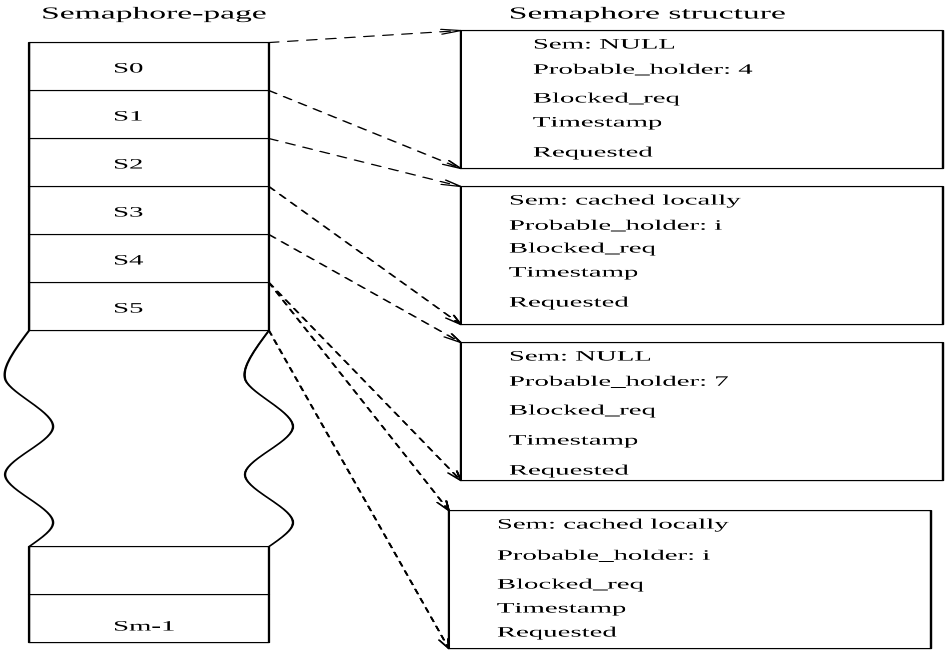

Figure 1: State of semaphore-page SP_j on processor P_i

As an example, Figure 1 depicts the state of four semaphore structures (S_0, S_2, S_4 and S_5) in semaphore-page, SP_j on processor P_i. Semaphores S_2 and S_5 are cached locally and semaphores S_0 and S_4 are on processors P_4 and P_7, respectively. The blocked_req queue, timestamp and requested fields are left blank.

3.2 P Operation

When a process tries to perform a **P** operation, it first checks the local semaphore-page on its processor for presence of the semaphore. If the semaphore is found locally, the **P** operation is performed. Otherwise, the probable_holder field indicates where the semaphore is likely to be cached and by using the probable_holder fields on successive processors, the current holder of the semaphore can be located. Thus, to acquire the semaphore a request is sent to the processor specified in the probable_holder and the **P** operation remains blocked until the semaphore has been cached locally. At this point the semaphore can be successfully decremented to complete the **P** operation.

A binary semaphore is not be moved to the requesting processor if it has a pending **V** operation on the processor that currently holds it. In the case of a resource counting semaphore, it is not moved from the processor where it is held unless it has a value greater than zero. Subsequent **V** operations are forwarded to the processor where the semaphore has moved. **V** operations are discussed in detail in Section 3.3.

A request message contains the id or tag of the semaphore being requested as well as the ID of the processor that initiated the request, (eg. [REQUEST, $<SP_i,S_j>$, P_k]). When the request is satisfied, the semaphore's probable_holder field must be updated to reflect the current location of the semaphore. The code that is executed on a processor to execute a **P** ($<SP_i,S_j>$) operation is shown in Figure 2. It is executed atomically as are all the subsequent code.

```
<<if (<SPi,Sj>.probable_holder != My_ProcId) then
   {
    if (<SPi,Sj>.requested != TRUE) then
     {
       SEND [REQUEST, <SPi,Sj>, My_ProcId] to <SPi,Sj>.probable_holder;
       <SPi,Sj>.requested := TRUE;
       }
     await reply;
     <SPi,Sj>.Sem := semaphore received in reply;
     <SPi,Sj>.probable_holder := My_ProcId;
     <SPi,Sj>.requested := FALSE;
     concat({blocked_req queue from sender}, <SPi,Sj>.blocked_req);
   }
  if (<SPi,Sj>.Sem >= 1) then <SPi,Sj>.Sem := <SPi,Sj>.Sem - 1;
    else  append(<SPi,Sj>.blocked_req, My_ProcId); >>
```

Figure 2: Code Executed to Access a Semaphore and Perform a **P** Operation

3.2.1 Servicing a Remote Access Request for a Semaphore

When a processor recieves a request for a semaphore, it checks if the semaphore is locally cached. If the semaphore is cached, it is sent to the requesting processor provided the semaphore's value is greater than zero. If the semaphore is not cached, the request is forwarded to the processor indicated by the probable_holder field. If a processor waiting to access a semaphore receives a request for the same semaphore from another processor, it will queue the request on the blocked_req queue for that semaphore.

It is possible that more than one request is queued for a semaphore on a processor. When the semaphore is released, it is sent to the processor of the first blocked request along with the blocked_req queue of other blocked requests. An alternative solution is to forward a request to the processor whose request arrived last (i.e., the request at the tail of the blocked_req queue) to be queued there. However, this distributed queue approach will increase the number of messages in the system.

The code to service a request of the form [REQUEST, $<SP_i,S_j>,P_k$] is as follows:

```
<<  if (<SPi,Sj>.probable_holder = My_ProcId) then
      {
       if (<SPi,Sj>.Sem >= 1) then
        {
         SEND [REPLY, <SPi,Sj>, <SPi,Sj>.Sem>] to Pk;
         <SPi,Sj>.probable_holder := Pk;
         <SPi,Sj>.Sem := NULL;
         }
       else
         append(<SPi,Sj>.blocked_req , Pk);
     }
  else
    {
      if (<SPi,Sj>.requested != TRUE) then
       {
         SEND [REQUEST, <SPi,Sj>, Pk] to <SPi,Sj>.probable_holder;
         <SPi,Sj>.probable_holder := Pk;
       }
      else
        append(<SPi,Sj>.blocked_req , Pk);
    } >>
```

Figure 3: Code to service a request for a semaphore

It is noteworthy that the manner in which a site responds to a request does not depend on whether the semaphore is binary or resource counting. The initial value that was assigned to the semaphore will determine the type of the semaphore. This, of course, assumes that every process that performs a **P** will eventually perform a **V** as well.

3.3 V Operation

When a process performs a **V** operation on a semaphore, two possibilities arise. First, the **V** operation is on a binary semaphore. In this case the operation can always be performed locally because the semaphore could not have been moved when it had a **V** operation pending. Second, the **V** operation is on a resource counting semaphore. In this case, it is possible that the semaphore has been moved to another processor and in such a case, the **V** operation must also be forwarded to that processor. As in the case of the semaphore access request, it may be necessary to forward the **V** operation until it reaches the current holder of the semaphore. The processor that holds the semaphore may have requests waiting on the blocked_req queue of the semaphore. In this case, after performing the **V** operation, the processor sends the semaphore to the first request on the blocked_req queue along with any other blocked requests.

A processor which performs the **V** operation in response to a request by a local process as well as one that performs the **V** operation in response to a remote operation request executes the same instructions (statements 1–10) in the code (see Figure 4). Processors that initiate a remote **V** operation message as well as those that simply forward such operation messages execute the else case (statement 13). The **V** ($<SP_i, S_j>$) operation is as follows:

```
<<if (<SPi,Sj>.probable_holder = My_ProcId) then
   {
    <SPi,Sj>.Sem = <SPi,Sj>.Sem + 1;
    if (<SPi,Sj>.blocked_req  != NULL) then
      {
        <SPi,Sj>.probable_holder = head(<SPi,Sj>.blocked_req );
        SEND [REPLY, <SPi,Sj>, <SPi,Sj>.Sem, tail(<SPi,Sj>.blocked_req )]
            to  head(<SPi,Sj>.blocked_req );
        <SPi,Sj>.blocked_req  = NIL;
        <SPi,Sj>.Sem = NULL;
       }
    }
  else
     SEND [VOP, <SPi,Sj>] to <SPi,Sj>.probable_holder;>>
```

Figure 4: Code to perform a **V** operation.

3.4 Updating the Structures in the Semaphore-page

We now discuss how the semaphore pages are updated. This is done when messages are exchanged between the processors when accessing a semaphore or performing a **V** operation. The timestamp field of the semaphore structures play an important role in this process. This field indicates how current the probable_holder information in a particular semaphore structure is compared to the rest of the system. By comparing the timestamps of a particular semaphore in two semaphore-pages, we can determine which semaphore-page has the more uptodate information about the semaphore's location. This is used to update the older of the two semaphore structures. The motivation behind updating the semaphore structure is that on average it would shorten the length of the path taken by a request to access a semaphore.

When a processor requests another processor for a semaphore, it also sends the probable holder information (details in Section 3.4.1) of the other semaphores of that semaphore-page as part of the request message. This is used by the other processors to update their semaphore-pages. As the request is forwarded to the holder, the processors along the path participate in updating their respective semaphore-pages. The requesting processor eventually updates its semaphore-page when it receives the information along with the semaphore from the processor which holds the semaphore.

As an example, suppose processor P_1 sends a request for $<SP_i, S_j>$ to P_2, which forwards it to P_3. P_1 sends probable holders information about the semaphores of the semaphore-page SP_i to P_2. Processor P_2 uses this information to update its copy of SP_i and then sends this updated version of the probable holder information of SP_i to P_3. The processor P_3 similarly updates its copy of SP_i. When it sends the semaphore to P_1, it will also send the SP_i probable holder information. This enables P_1 also to update its own copy of SP_i. In this manner, a processor gets the holder information from the processor prior to it in the path and passes its updated information to the successive processor in the path.

3.4.1 Specifics of the Update Scheme

As part of the messages, an array of tuples, called *holder_status* is now included. This array of tuples has m elements; one tuple per semaphore of a semaphore-page. Each tuple consists of the probable_holder and timestamp fields (see Figure 5) that were described in the semaphore structure in Section 3.1. This array is created from the semaphore-page prior to issuing any message.

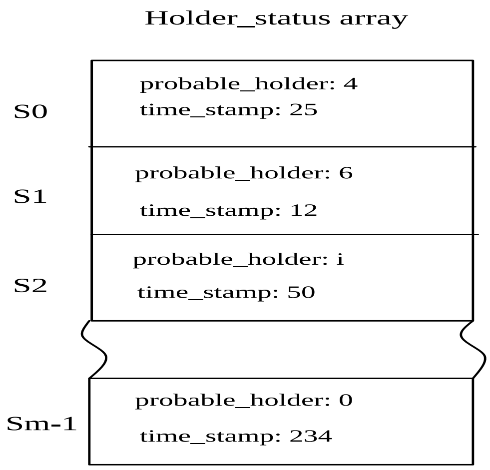

Figure 5: Holder_status array for semaphore-page SP_j on processor P_i

A processor that wishes to access a semaphore that it not available locally, issues the following request message: ([REQUEST,<SP_i,S_j>,{holder_status of SP_i},P_k]. The processor then waits for the semaphore and the holder_status array of the page SP_i from the holder of the semaphore. It uses the holder_status array to update the other semaphores of its semaphore-page. It also increments the timestamp of the semaphore it received. This ensures that the timestamp at the holder of a semaphore is the largest of all the timestamps for that semaphore in the system. The code from Section 3.2 is modified to reflect this change:

When a processor receives a message, it uses the holder_status array sent in the message to determine which semaphore structures in its semaphore-page to update. It determines which probable_holder entry is more recent by comparing the timestamps supplied in each tuple of the holder_status array with those in the semaphore structures of SP_i. The code to update the local semaphore-page SP_i is given in Figure 7.

```
<< if (<SPi,Sj>.probable_holder != My_ProcId) then
    {
      if (<SPi,Sj>.requested !=  TRUE) then
       {
        SEND [REQUEST, <SPi,Sj>, {holder_status of SPi}, My_ProcId]
         to <SPi,Sj>.probable_holder;
      <SPi,Sj>.requested := TRUE;
      await reply;
      Update SPi using the holder_status array received with the reply.
      <SPi,Sj>.Sem := semaphore received in reply;
      <SPi,Sj>.probable_holder := My_ProcId;
      <SPi,Sj>.timestamp := <SPi,Sj>.timestamp + 1;
      <SPi,Sj>.requested := FALSE;
      concat({blocked_req queue from sender}, <SPi,Sj>.blocked_req);
    }
   if (<SPi,Sj>.Sem >= 1) then <SPi,Sj>.Sem := <SPi,Sj>.Sem - 1;
     else  append(<SPi,Sj>.blocked_req, My_ProcId); >>
```

Figure 6: Modified code to acccess a semaphore.

When a processor has to forward a request message, it first updates its own semaphore-page using the holder_status array. It then creates a new holder_status array based on its semaphore-page to send with the forwarded message. The modified code to service a request for a semaphore is given in Figure 8.

When a processor performs a **V** operation that requires it to send a message to

```
<<for (index := 0 to m-1) do
    if (info_struct[index].time_stamp > <SPi,Sj>.time_stamp) then
      {
       <SPi,Sj>.probable_holder := info_struct[index].probable_holder;
       <SPi,Sj>.time_stamp := info_struct[index].time_stamp;
      }>>
```

Figure 7: Code to update the semaphore-page.

```
<<  Update SPi using the holder_status array received.
    if (<SPi,Sj>.probable_holder = My_ProcId) then
     {
      if (<SPi,Sj>.Sem >= 1) then
       {
          <SPi,Sj>.probable_holder := Pk;
          <SPi,Sj>.timestamp := SPi,Sj>.timestamp + 1;
          SEND [REPLY, <SPi,Sj>, <SPi,Sj>.Sem, {holder_status of SPi}]
              to Pk;
          <SPi,Sj>.Sem := NULL;
        }
       else
          append(<SPi,Sj>.blocked_req , Pk);
     }
    else
     {
        if (<SPi,Sj>.requested != TRUE) then
          SEND [REQUEST, <SPi,Sj>, {holder_status of SPi}, Pk]
              to <SPi,Sj>.probable_holder;
          <SPi,Sj>.probable_holder := Pk;
        }
        else
          append(<SPi,Sj>.blocked_req , Pk);
      } >>
```

Figure 8: Modified code to service a request for a semaphore.

another processor (i.e., a resource counting semaphore which has been moved), it includes the holder_status array of the semaphore-page in the **V** message. So, the message structure is: ([VOP,$<SP_i$,$S_j>$,{holder_status of SP_i}].

This enables the processor performing the **V** operation to update its semaphore-page when it performs the **V** on the semaphore. If it is necessary to forward the **V** operation message via a chain of processors, the processors along the path update their respective semaphore-pages prior to forwarding the **V** message. The code below is a modified version of the one presented in Section 3.3. While, the version presented earlier was uniform for both local and remote **V** operations, it is necessary to differentiate between them now. The code below in Figure 9 is for a remote **V** operation. If a processor is able to perform a **V** operation locally, it will execute all the statements in Figure 9 except statement 1.

```
 1.<< Update semaphore page SPi using the information structure;
 2.   if (<SPi,Sj>.probable_holder = My_ProcId) then
 3.    {
 4.     <SPi,Sj>.Sem = <SPi,Sj>.Sem + 1;
 5.     if (<SPi,Sj>.blocked_req  != NIL) then
 6.      {
 7.        <SPi,Sj>.probable_holder = head(<SPi,Sj>.blocked_req );
 8.        <SPi,Sj>.timestamp = <SPi,Sj>,timestamp + 1;
 9.        SEND [REPLY, <SPi,Sj>, <SPi,Sj>.Sem, tail(<SPi,Sj>.blocked_req ),
               {holder_status of SPi}] to  head(<SPi,Sj>.blocked_req );
10.        <SPi,Sj>.blocked_req  = NIL;
11.        <SPi,Sj>.Sem = NULL;
12.       }
13.     }
14.   else
15.     SEND [VOP, <SPi,Sj>,{holder_status of SPi}] to
             <SPi,Sj>.probable_holder; >>
```

Figure 9: Modified code to perform a **V** operation.

3.5 Correctness

In this section, we present the correctness of the scheme. We show that two properties hold. First, we show that the **P** and **V** operations are performed correctly (i.e., the saftey property is satisfied). Second, we show that a requested **P** or **V** operation is satisfied in finite time (i.e., the liveness property is satisfied). We show that these properties hold for one semaphore and the same argument applies to other semaphores.

3.5.1 Safety Property

Lemma 1 *Only one copy of the semaphore exists in the system.*

Proof: Initially, only one processor holds the semaphore and all the other processors have the value NULL for it in their semaphore-page. When the semaphore is forwarded by the holder processor, it sets the corresponding $<SP_i,S_j>$.Sem field to NULL (in Statement 9 of Figure 8 and in Statement 11 of Figure 9). When a processor received it, it places the semaphore variable in its $<SP_i,S_j>$.Sem field (Statement 9 in Figure 6). □

Theorem 1 *The* **P** *and* **V** *operations are performed correctly.*

Proof: There are two cases to consider. First, that the **P** and **V** operations are performed in a mutually exclusive manner. Second, they are performed according to their definitions (stated in Section 3.1).

Case 1: From Lemma 1, only one processor can hold the semaphore at a time; therefore, only one processor will perform **P** or **V** on the semaphore at any time. Nonetheless, several processes at a processor may concurrently perform **P** and **V** operations on the semaphore. These concurrent operations on the semaphore structure are serialized in critical sections. Hence, only one operation (either a **P** or a **V** operation) can be done at a time on the semaphore.

Case 2: As per the definition, a **P** operation will succeed if the semaphore's value is greater than zero. Otherwise, the request is blocked. This is ensured in our algorithm in Figure 6. A **V** operation is performed by the processor holding the semaphore (not necessarily the one requesting it). This processor increments the semaphore and if there are any blocked requests, it passes the semaphore to that processor, which is equivalent to unblocking a request on that processor. □

3.5.2 Liveness Property

Observation 1 *The directed graph induced by the probable_holder fields of a semaphore is a spanning forest.*

This observation holds at the very beginning of the computation, when the probable_holder fields on all the processors contain the processor ID of the semaphore holder. The field is modified when a processor passes the semaphore to an processor which has requested access to the semaphore. When a request passes through a processor, the probable_holder field on that intermediate processor is set to the requesting processor. The requesting processor will not make another request until it receives and subsequently gives up the semaphore. These requesting processors form roots of their respective trees.

Observation 2 *A processor's request will never loop back to the originating processor.*

If processor P_1 requested processor P_2 for a semaphore, it will result in P_2 pointing to P_1 as the probable_holder. This is true irrespective of whether P_2 held the semaphore or simply forwarded the request. Before P_1 requests P_2 again, the semaphore must have been cached on P_1 and then forwarded. But, unless P_1 forwarded the semaphore to P_2 last time, the next request could not have arrived P_2. Thus, either this second request from P_1 never gets to P_2 or if it does, P_2 will either have the semaphore or would have forwarded the semaphore to a processor other than P_1 at that time.

Lemma 2 *A request message will reach the semaphore holder in finite steps.*

Proof: The are two possibilities: either the request message reaches the holder of the semaphore or it is queued up at a processor which itself is requesting the semaphore.

A processor holding the semaphore will have its probable_holder field point to itself. If a semaphore is not available locally, the probable_holder field for that semaphore specifies the processor where the semaphore was forwarded. If that processor has itself forwarded the semaphore to another processor, its probable_holder field will point to that processor, and so on. The semaphore request travels through all these processors before it reaches the holder processor. In the worst case, the semaphore request will be forwarded through $n-2$ processors, resulting in a path of length $n-1$ to get to the holder of the semaphore.

If the request reaches a processor which has an outstanding request for the semaphore (Request = TRUE), then the request is queued at that processor. By applying this case recursively, a chain of blocked requests, each on a different processor is created. The processor at the head of this chain would have a request which reaches the semaphore holder via a path whose length between 1 and $n-2$. $\square$

Lemma 3 *A blocked request will be granted the semaphore in finite time.*

Proof: A request that can not be granted the semaphore is queued on the blocked_req queue at a processor. If the request is at the head of the queue, it will receive the semaphore, when it can be forwarded. If it is at the i^{th} position in the queue, after the queue has been forwarded to the processor at the head of the queue, the request is the now at $(i-1)^{th}$ position in the queue. At every intermediate processor, the local queue is appended to the current queue. After the queue has been forwarded $i-1$ times, the request is at the head of the queue and is granted the semaphore next. □

Theorem 2 *Every request for a semaphore is eventually satisfied.*

Proof:The proof of this theorem is a direct result of lemmas 2 and 3. □

3.6 Performance

The performance is determined by the message traffic generated and other properties such as reliability. While a centralized scheme uses the fewest messages to perform a **P** or **V** operation on a semaphore, it also creates a bottleneck at that processor. Furthermore, it suffers from the problem of a single point of failure. Distributed schemes such as Munin's locks would require on average $O(log\ n)$ number of messages to find the current holder of the lock. If there are multiple requests (say k), the request are not queued up at the holder, but instead are chained at in a distributed queue. This requires that each request is forwarded to the last requested processor to be queued there. This results in $k-1$ additional messages.

The graph induced by the probable_holders in our system will typically be a fat tree and the average number of messages required to get a request queued at the semaphore holder is $O(log\ n)$. However, if there are a number of processors requesting, a request message is likely to be queued up at one of the requesting processors. This would reduce the number of messages before a request is queued (at one of the requesting processors). Under a high request pattern, the average number of messages per request would be significantly less than $O(log\ n)$. If the request pattern is sparse, the graph can become rather thin and in the worst case the number of messages needed to get a request on a queue would be $n - 1$.

4 Conclusion

Semaphores are an intrinsic part of concurrent programming in shared memory systems and thus should be provided in distributed shared memory systems. In this paper we have presented an approach to providing semaphores on a distributed shared memory systems. Our approach differs from previous work in that it decentralizes the management of the semaphores. This decentralization eliminates the inherent problems in centralized schemes: the single point of failure and the bottlneck problems.

By grouping semaphores into semaphore-pages, our approach enables processes to exploit the locality of reference which exists in accessing semaphore variables by caching them on their processors. By updating information about the locations of all the semaphores in a semaphore-page, the semaphore location algorithms becomes simple yet efficient.

Future research involves a comprehensive performance study of various synchronization mechanisms. Also, fault tolerance aspects of these mechanisms need to be studied.

References

[1] Kai Li and Paul Hudak. "Memory Coherence in Shared Virtual Memory Systems". *ACM Transactions on Computer Systems*, 7(4):321–359, November 1989.

[2] Bill Nitzberg and Virginia Lo. "Distributed Shared Memory: A Survey of Issues and Algorithms". *IEEE Computer*, pages 52–60, August 1991.

[3] John Bennett, John Carter, and Willy Zwaenepoel. "Munin: Distributed Shared Memory Based on Type-Specific Memory Coherence". In *Symp. on Principles and Practice of Parallel Programming*, pages 168–175, March 1990.

[4] P. Dasgupta, R. LeBlanc, and W. Appelbe. "The Clouds Operating System: Functional Description, Implementation details and Related Work". In *International Conference on Distributed Computing Systems*, pages 2–9, June 88.

[5] Roberto Bisiani and Mosur Ravishankar. "Plus: A Distributed Shared-Memory System". In *International Symp. on Computer Architectures*, pages 115–124, May 1990.

[6] Ming-Chit Tam, Johnathan Smith, and David Farber. "A Taxonomy-Based Comparison of Several Distributed Shared Memory Systems". *ACM-Operating Systems Review*, 24(3):40–67, July 1990.

[7] U. Ramachandran and M. Y. A. Khalidi. An Implementation of Distributed Shared Memory. *Software - Practice and Experience*, 21(5):443–464, May 1991.

[8] John Carter, John Bennett, and Willy Zwaenepoel. "Implementation and Performance of Munin". In *ACM Symp. on Operating System Principles*, pages 152–164, October 1991.

[9] Roberto Bisiani, Andreas Nowatzyk, and Mosur Ravishankar. Coherent shared memory on a distributed memory machine. In *International Conference on Parallel Processing*, pages I133–I141, 1989.

[10] Kai Li and Richard Schaefer. A Hypercube Shared Virtual Memory System. In *International Conference on Parallel Processing*, pages I125–I132, 1989.