Deadlock Detection Without Wait-For Graphs

Dror G. Feitelson
Department of Computer Science
The Hebrew University of Jerusalem
91904 Jerusalem, Israel

Abstract

Deadlock detection is an important service that the run-time system of a parallel environment should provide. In parallel programs deadlock can occur when the different processes are waiting for various events, as opposed to concurrent systems, where deadlock occurs when processes wait for resources held by other processes. Therefore classical deadlock detection techniques such as checking for cycles in the wait-for graph are unapplicable. An alternative algorithm that checks whether all the processes are blocked is presented. This algorithm deals with situations in which the state transition from blocked to unblocked is indirect, as may happen when busy-waiting is used.

1 Introduction

A major problem encountered by parallel programming novices, which does not exist in sequential programming, is that bugs may cause parallel programs to deadlock. It is therefore important for the run-time system to identify deadlock situations and to report them to the user.

Deadlock detection has been studied extensively in the context of concurrent (time sharing) uniprocessor systems. Deadlock can occur in such systems due to cyclic patterns of requests for exclusive access to system resources. Algorithms for deadlock prevention, detection, and recovery are standard material in operating systems textbooks (e.g. [8, 7]). Such algorithms have also been extended to distributed systems, where competing processes may execute on different processors. The central issue in these extensions is the distributed maintenance of the *wait-for graph*, that shows which process is waiting for a resource held by which other process [10, 9, 4]. Deadlock detection is reduced to finding cycles in this graph.

The approach of maintaining a wait-for graph and searching for cycles in it does not generalize to parallel programming. The reason is that in a parallel program it is possible for a process to wait for an event, without knowing which other process will cause that event to happen. Thus it is impossible to maintain a wait-for graph in the first place. However, there is also a bright side. As all the processes in a given application cooperate to achieve the same goal, the application is only considered deadlocked if *all* of them are waiting. Hence it is possible to detect deadlock by simply checking if none of the processes is in a runnable state.

The only problem with this approach is that indirect state changes do not necessarily take effect immediately. For example, this may happen if busy-waiting is used to delay the process that is waiting for an event. Thus when process $\mathcal{A}$ wakes suspended process $\mathcal{B}$ it is actually only setting the value of a certain shared variable. If process $\mathcal{A}$ subsequently suspends itself, there may be a certain time when both are suspended because process $\mathcal{B}$ did not check the variable yet. This could lead to false alarms. A global computation is needed to ascertain that indeed all the processes are

waiting and none are about to resume execution. The idea is to do this independently of the actual shared variables used by the application to implement busy-waiting. In other words, the run-time system should not have to keep track of all the different events and what process is waiting for which event.

The rest of the paper is organized as follows. Section 2 specifies the context of our work by detailing the system model and assumptions. The deadlock detection algorithm is presented in section 3. A couple of examples of how this algorithm is incorporated in the implementation of synchronization primitives are given in section 4. The conclusions are drawn in section 5.

2 System Model and Assumptions

The system under consideration is a tightly-coupled shared-memory multiprocessor, that supports fine-grain parallel computations. Bus-based systems such as the Sequent Balance and Encore Multimax are possible examples. These systems encourage the use of shared memory to coordinate the activities of processes. When one process needs to wait for another, busy-waiting is often used [1]. If more than one process is mapped to each processor, the busy-waiting loop may include an instruction to yield the processor to another ready process. This prevents computation cycles from being wasted without requiring extensive bookkeeping to keep track of which process is waiting for what event. It is important to note that a process that is waiting for an event is not suspended from execution — it can still be scheduled to run, but the run-time system knows that it is busy waiting.

Shared memory machines typically support some sort of atomic operations in hardware, thus enabling the users to devise various synchronization schemes. Our deadlock detection algorithm requires the ability to set and reset a bit in a shared memory word atomically. The implementations described in section 4 require atomic lock and fetch-and-add operations [6]. Implementations based on other primitives are also possible, but slightly more complicated.

To simplify the presentation, it is also assumed that processes are mapped to processors upon creation and do not migrate at run time. This assumption is reasonable in the context of parallel processing, where load-balancing is less beneficial than in distributed systems [5]. However, this assumption is not required and the algorithm is easily extended to deal with migration. In addition it is assumed that only one application is running at a time, and all the processes belong to it. This assumption is also easily removed by performing the necessary bookkeeping for each application independently.

Finally, it is assumed that the run time system is designed in a distributed style, with most of the data structures maintained locally on each processor independently. This assumption is in line with the assumption that processes only execute on one processor. However, it does not preclude the use of shared memory by the run time system when necessary.

3 Deadlock Detection Algorithm

By definition, deadlock occurs if and only if all the application processes are stuck. Therefore deadlock may be detected by maintaining a count of the total number of processes, and a count of the stuck processes, and comparing the two.

As processes always execute on the same processor, it is easy to maintain both of these counters. The `total` counter is incremented on process creation and decremented on process termination.

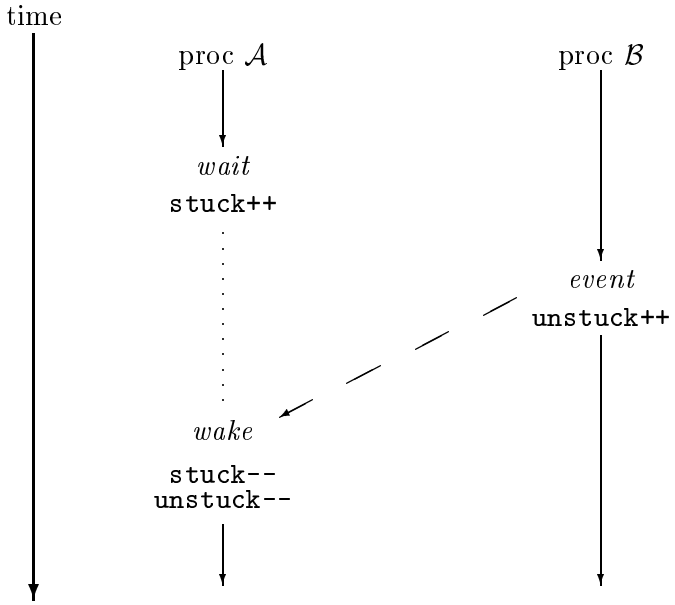

Figure 1: *Maintaining the* `stuck` *and* `unstuck` *counters.*

The `stuck` counter is incremented when a process enters a busy waiting loop and decremented when it exits from the loop. If the counters are equal, the run time system knows that all the local processes are stuck. As the processor cannot do useful work when all the processes are stuck, the run time system can initiate a check for deadlock without compromising the overall performance. The check consists of verifying that all the processes on the other processors are also stuck.

Regrettably, the algorithm as presented so far is incorrect in that it may cause false alarms, reporting a deadlock situation when actually there is no deadlock. The problem arises from the fact that processes become "unstuck" implicitly when some shared variable is set to some value, but the `stuck` counter is only updated when the process is scheduled and checks the variable. Thus a situation in which the counters show that all the processes are stuck does not necessarily imply deadlock, because the counters may change when processes are scheduled.

The problem is solved by adding a third counter, called `unstuck`. This counter is used by each processor to maintain its net contribution to system-wide releasing of processes from the stuck state. It is incremented whenever an event that releases some process occurs, and decremented whenever a process notices that an event occurred. This protocol is shown schematically in fig. 1. Note that the process that causes the event may execute on a different processor than the process that is waiting for it. In this case the `unstuck` counter on one processor is incremented, and the counter on another processor is decremented.

The algorithm may be summarized as follows:

1. The run time system maintains three local counters on each processor:
   (a) `total` is incremented on process creation and decremented on process termination,
   (b) `stuck` is incremented when a process begins to wait and decremented when it notices that the awaited event occurred, and
   (c) `unstuck` is incremented when an event is caused and decremented together with `stuck`.

2. Whenever `total` = `stuck`, a shared bit indicating that this processor has nothing to do is set. If the counters change, the bit is reset. Efficiency and stability may be improved by periodic checking for equality, rather than every time that the counters are changed. However, this is just an implementation detail.

3. A processor that finds all the bits set initiates a synchronous calculation of global state by summing all the local values of `unstuck`. If the number of processors is small enough so that all the bits are in one word, they can all be read together. Otherwise it is necessary to read a number of words. In either case, each processor verifies that `total` = `stuck` when it adds its local value of `unstuck` to the global sum. If some processor finds that `total` $\neq$ `stuck`, the algorithm is aborted. If the sum is zero, deadlock is announced.

Correctness of this algorithm is established by the following sequence of claims. P denotes the number of processors in the system, and the counter values on the ith processor are denoted by $\texttt{total}_i$, $\texttt{stuck}_i$, and $\texttt{unstuck}_i$.

Claim 1 *When deadlock is suspected, i.e.* $\forall i : \texttt{stuck}_i = \texttt{total}_i$*, the following inequality holds:*

$$\sum_{i=1}^{P} \texttt{unstuck}_i \;\geq\; 0.$$

This is so because when each event occurs it first causes the counter on the processor where it occurred to increment, and only later when it is noticed it causes the counter on another processor to decrement. It is true that in general the decrement may take effect before the increment. However, as long as the process that caused the event does not increment `unstuck`, it is itself active. Therefore on its processor `stuck` < `total`, and we do not suspect deadlock.

The sum of the `unstuck` counters represents the number of processes that should be considered as unstuck, but are not reflected yet in decrementing the `stuck` counters. Therefore we have

Claim 2 *The number of processes that are* really *stuck is*

$$\sum_{i=1}^{P} (\texttt{stuck}_i - \texttt{unstuck}_i).$$

Deadlock occurs if and only if all the processes are really stuck, which is expressed as

$$\sum_{i=1}^{P} \texttt{total}_i = \sum_{i=1}^{P} (\texttt{stuck}_i - \texttt{unstuck}_i).$$

As it is obvious that $\forall i : \texttt{stuck}_i \leq \texttt{total}_i$, and in fact when deadlock is suspected we know that $\forall i : \texttt{stuck}_i = \texttt{total}_i$, equality can hold only if the sum of the `unstuck` counters is zero. Thus we have

Claim 3 *A necessary and sufficient condition for deadlock is*

$$\forall i : \texttt{stuck}_i = \texttt{total}_i \quad \text{and} \quad \sum_{i=1}^{P} \texttt{unstuck}_i \;=\; 0.$$

```
/* barrier is initialized to the number of processes n */
/* flag is initialized to zero */

if ( F&A( barrier, -1 ) == 1 ) { /* last to arrive */
   flag = 1;
   F&A( unstuck, n − 1 );
}
else { /* wait for someone to set the flag */
   F&A( stuck, 1 );
   while (!flag)
      yield_processor;
   F&A( stuck, −1 );
   F&A( unstuck, −1 );
}
```

Figure 2: *Implementation of barrier synchronization in C-like code.* `barrier` *and* `flag` *are shared.* `F&A` *is an atomic fetch-and-add operation.*

Note that this condition has a local part and a global part. For efficiency reasons, the algorithm checks the global sum only when the local condition $\texttt{stuck}_i = \texttt{total}_i$ is satisfied on all the processors.

The cost of the algorithm is negligible. During normal operation, all the bookkeeping is done in a distributed and asynchronous manner, with no interaction between the processors. As long as there is any active process on the processor, the bookkeeping only consists of incrementing and decrementing local counters when events occur, so the overhead is low. The computation of global state is also simple — all that is needed is to sum a set of counters, one per processor. Even this is only done when all the processors have reason to believe that all their processes are not doing useful work, but rather busy waiting.

4 Implementation of Synchronization Primitives

To illustrate the algorithm, consider the following implementations of barrier synchronization and semaphores. These examples show how the counters are maintained — the global summation is not included. As the counters are used by all the processes that time-share on the same processor, they are updated using an atomic fetch-and-add operation (denoted `F&A`).

The code for barrier synchronization is given in fig. 2. This is a straightforward use of busy-waiting on a shared flag until all the processes arrive. The last process to arrive releases $n - 1$ waiting processes, so it increments the `unstuck` counter by this amount. The instructions added for deadlock detection are emphasized by bold type.

Semaphores are interesting because they are used both for mutual exclusion and for event synchronization [3]. The implementation is given in fig. 3. This code is somewhat tricky, as it uses two distinct internal variables for the semaphore value: one is the value as it is seen by newly arriving processes that perform a `P` operation (`sem->val` in the figure), and the other is the value

```
struct {
   int     val;          /* semaphore value */
   int     waiting;      /* initially zero */
   int     released;     /* initially zero */
   lock_t  lock;         /* initially unlocked */
} semaphore;

P( sem )
semaphore *sem;
{
   lock(sem->lock);
   if ( sem->val > 0 ) { /* get in on first try */
      sem->val--;
      unlock(sem->lock);
      return;
   }
   else { /* fail: mark that I'm stuck */
      sem->waiting++;
      unlock(sem->lock);
      F&A( stuck, 1 );
      yield_processor;
   }

   /* here on second try and later */
   while (TRUE) {
      if ( sem->released == 0 ) { /* no luck yet */
         yield_processor;
         continue;
      }
      lock(sem->lock);
      if ( sem->released > 0 ) { /* got in finally */
         sem->released--;
         unlock(sem->lock);
         F&A( stuck, -1 ); /* mark that I'm OK again */
         F&A( unstuck, -1 );
         return;
      }
      else { /* missed this chance... */
         unlock(sem->lock);
         yield_processor;
         continue;
      }
   }
}
```

Figure 3: *Implementation of semaphores: definition and* `P` *operation.*

```
V( sem )
semaphore *sem;
{
   lock(sem->lock);
   if ( sem->waiting > 0 ) { /* need to release someone */
      sem->released++;
      sem->waiting--;
      unlock(sem->lock);
      F&A( unstuck, 1 ); /* mark that someone is free */
   }
   else { /* nobody is waiting */
      sem->val++;
      unlock(sem->lock);
   }
}
```

Figure 3 (cont.): *Implementation of semaphores:* `V` *operation.*

seen by processes that are waiting for it to become positive (`sem->released`). This distinction is needed to keep the counters used for deadlock detection consistent.

If only one variable is used, a process performing a `V` operation when some other process is waiting would increment the semaphore value and also increment the `unstuck` counter, expecting the waiting process to enter the semaphore. But if a new process now performs a `P` operation it may "steal" the semaphore without waiting, thus preventing the waiting process from registering the event and decrementing the `unstuck` counter. As a result the sum of the `unstuck` counters would always be positive, causing the deadlock detection algorithm to fail. To prevent this scenario, processes that busy-wait in the `P` operation indicate this be incrementing the `sem->waiting` counter. A process that performs a `V` operation and finds `sem->waiting` to be non-zero, releases one of the waiting processes by incrementing `sem->released` rather than `sem->val`.

5 Conclusions

The synchronization requirements of parallel programming are different from those which are typical in concurrent and distributed systems. For example, parallel programs often use event synchronization where is is unknown which process will cause the event. When the use of such synchronization leads to deadlock, it is impossible to attribute the deadlock to a cycle in a wait-for graph because there is no wait-for relation among processes. Therefore new deadlock detection algorithms are needed.

If events are used to change the state of a waiting process from blocked to ready directly, there is a simple solution; All that is needed is to check if all the processes are blocked at any given moment. If events effect processes indirectly, e.g. by setting the value of a shared variable that is subsequently checked by a busy-waiting process, the situation is somewhat more complicated. A double bookkeeping scheme was devised to deal with such behavior, where the run time system on

each processor counts the number of local blocked processes, and also counts its net contribution to the global number of blocked processes. This allows for a global state to be computed when deadlock is suspected.

The algorithm works if and only if it is guaranteed that the waiting processes will eventually notice that the event has occurred. This condition can typically be satisfied through careful coding. However, situations in which it cannot be satisfied do exist. For example this might happen if it is possible for one process to kill other processes. If a killed process was waiting for an event that had already occurred, but it had not noticed the event before being killed, the algorithm may fail to report a real deadlock situation. Thus deadlock detection is not possible if the system combines indirect transitions between blocked and unblocked states with the ability to kill other processes. The only solution to this situation is to use event-specific suspend and resume instructions, that explicitly tell the run-time system what event each process is waiting for. The run-time system must then perform its bookkeeping on a per-event basis.

The deadlock detection scheme reported in this paper was developed as part of the run time library for the *ParC* parallel language [2] implementation on the Makbilan research multiprocessor.

References

[1] T. E. Anderson, The performance of spin lock alternatives for shared-memory multiprocessors, *IEEE Trans. Parallel & Distributed Syst.* **1** (1990) 6–16.

[2] Y. Ben-Asher, D. G. Feitelson, and L. Rudolph, *ParC* — an extension of *C* for shared memory parallel processing, The Hebrew University of Jerusalem, submitted for publication (1990).

[3] E. W. Dijkstra, Co-operating sequential processes, in: F.Genuys, ed., *Programming Languages* (Academic Press, New-York, 1968) 43–112.

[4] A. K. Elmagarmid and A. K. Datta, Two-phase deadlock detection algorithm, *IEEE Trans. Comput.* **37** (1988) 1454–1458.

[5] D. G. Feitelson and L. Rudolph, Mapping and scheduling in a shared parallel environment using distributed hierarchical control, in: *Intl. Conf. Parallel Processing* (1990) I-1–I-8.

[6] A. Gottlieb, R. Grishman, C. P. Kruskal, K. P. McAuliffe, L. Rudolph, and M. Snir, The NYU Ultracomputer — designing an MIMD shared memory parallel computer, *IEEE Trans. Comput.* **C-32** (1983) 175–189.

[7] S. Krakowiak, *Principles of Operating Systems* (MIT Press, Cambridge, 1988).

[8] J. Peterson and A. Silberschatz, *Operating System Concepts* (Addison-Wesley, Reading, 1983).

[9] M. Roesler and W. A. Burkhard, Resolution of deadlocks in object-oriented distributed systems, *IEEE Trans. Comput.* **38** (1989) 1212–1224.

[10] M. Singhal, Deadlock detection in distributed systems, *Computer* **22** (Nov 1989) 37–48.