# Extending Java to Support Shared Resource Protection and Deadlock Detection in Threads Programming

*by*
Anita J. Van Engen
Michael K. Bradshaw
Nathan Oostendorp

## Abstract

Java threads enable programmers to write parallel programs very easily and conveniently. However, the current Java specification does not adequately provide for the protection of shared resources or for deadlock detection, two of the most common problems arising from parallel programs. The ability to solve these problems is crucial for Java concurrent programming. In this paper, we introduce and implement a model that can aid in writing parallel programs by providing shared resource protection and deadlock detection.

## Introduction

The increasing use of parallel systems and programs in many different fields has brought new attention to how threads cooperate and share data. Parallel programs are used in such critical and sensitive environments as database operation and manipulation, data collection and system maintenance tasks. With an increasing emphasis being placed on the World Wide Web and on platform-independent tools, Java is positioned to become the language of choice for many applications, including parallel programming. While it provides some solid facilities for basic threads programming, Java does not adequately provide for shared resource protection or have the ability to detect deadlock. Adequate Java support for these features would make Java threads even more effective for parallel programming.

Let us consider two simple applications that illustrate the need for these features. First, consider a bank with 2 ATM machines. Bob and Jane share an account with a current balance of $500, and they each make a different transaction at the exact same time. Bob wants to withdraw $100 and Jane wants to deposit $150. If this scenario proceeds, without provisions for data protection, the resulting balance will be either ($500 - 100) = $400 or ($500 + 150) = $650, both of which are incorrect. This, of course, does not happen, and the correct balance will include both transactions, leaving the final balance at ($500 - 100 + 150) = $550. Second, consider the Dining Philosophers scenario [1] -- a classic resource sharing problem. There are five philosophers sitting at a table. Each has one fork between him and his neighbor on either side, for a total of five forks. Each philosopher needs two forks to eat, and can only pick up one fork at a time. The philosophers need to avoid "starvation," or the unfair sharing of resources. A problem arises if all five philosophers reach for their right forks at the same time. Each philosopher will have only one fork in their hand, and thus cannot eat. However, there are not any other forks on the table to reach for. The result of this situation is deadlock. Resource corruption and deadlock are common problems in parallel programming that are extremely difficult to detect.

These examples illustrate the need for resource protection in parallel programming, especially the need to prevent corrupted data and deadlock. In this paper, we will discuss the current ways in which Java is used for parallel programming and its associated problems. We will then introduce a model for Java resource protection. We will expand on this model by showing our implementation and performing some data analysis to demonstrate the effectiveness of our design. Finally, we will discuss implications of this model.

## Background

One strong point of the Java programming language is the inclusion of threads in its basic API [4]. The **Thread** class allows a programmer to write concurrent code quickly and easily. Currently, the method Java uses for concurrency control is to make a class' critical procedures ``synchronized." Only one thread is allowed to run a synchronized procedure at a time, and all other threads are queued. When execution of the current thread completes, or when it calls the **wait()** method, another thread is allowed to call a synchronized procedure. The **wait()** method will release the lock on the current synchronized procedure and cause the thread to sleep until the **notify()** procedure is called. While this ``monitor" based approach, introduced by Hoare [2], is a workable solution, placement of synchronization within procedures is often a difficult task for programmers trained only in writing serial procedures. Synchronized methods can also cause deadlock when used incorrectly, and the Java virtual machine has no native means of detecting a deadlock situation.

Consider our two previously mentioned examples. The ATM problem could easily be solved by synchronizing the methods that accessed the account. However, in a scenario where many people have access to the same account, this method can be too constrictive, and does not lend itself to more innovative, efficient designs. The dining philosophers problem can also be solved with synchronized methods, by synchronizing parts where the philosophers request and return forks. However, to solve the philosophers' problem one has to write a locking mechanism for the forks, as well as implement some sort of algorithm that will prevent the philosophers from entering a deadlock. These same considerations have to go into almost any resource-locking algorithm and could be more easily handled by built-in software controls.

The only known alternative to synchronized methods is a commercial package by Sealevel Software (http://www.halcyon.com/sealevel) called the Single-Write-Multiple-Read-Guard (SWMRG) [3]. (For information on SWMRG and a threads demo, see http://www.halcyon.com/sealevel/testswmrg.htm) It allows the programmer to protect data conservatively by allowing only one writer or multiple readers to have access to the same SWMRG object. However, it does not protect against a deadlock situation. Also, the programmer is expected to integrate the Singer-Write-Multiple-Read-Guard [3] into their own data structure, rather than having an Object protect itself, which seems more appropriate to the Java object-oriented style. Synchronized methods and Sealevel Software's SWMRG at this time are the only tools available to a parallel programmer who wants to protect data; neither of which handle the deadlock problem.

## The DynAC Model

Our model for resource protection is called the **DynAC Model**, which stands for Dynamic Anomaly Correction. DynAC is composed of five sections, as shown in Figure 1. The interesting features of the model are the interchangeability of components and the ability to monitor activity. This section will step through this model as an introduction to its implementation.

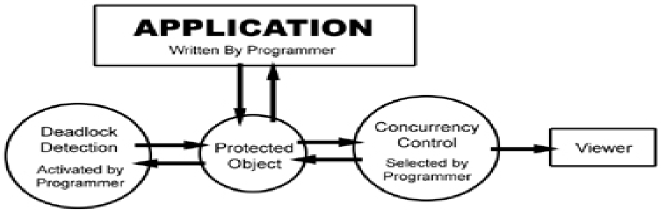


Figure 1: The DynAC Model

The **Application** is a parallel system that is written by the programmer. It requests permission to read, write, and update logical portions of the shared resources. It is the responsibility of the programmer to catch exceptions which may arise from a poorly designed concurrency control for the application. Because of the modular design of the DynAC Model, the same application may be used with different forms of concurrency control.

Shared data are protected through the use of the **ProtectedObject** class, an extension of the Java **Object** Class. Each logically-distinct unit of data is a ProtectedObject. For example, in the ATM problem each account would be a ProtectedObject. All requests to read, write or update the shared data are passed through the ProtectedObjects. Once a request is made, a ProtectedObject makes a call to its associated ConcurrencyControl module in order to determine if the threads are permitted to interact. Then the ProtectedObject determines the correctness of granting a request by using internal algorithms. When the application's request is processed, the thread is allowed to proceed if there are no complications. If an error occurs due to corrupted data or a deadlock of the resources, an exception is thrown for the application to deal with or to allow the program to close without corrupting the data.

**DeadLockDetection** is an optional feature that allows the ProtectedObject to evaluate the correctness of the ConcurrencyControl's permissions. In several ConcurrencyControls, a set of threads will stall if they are holding resources that other threads need and requesting the resources that the other threads are holding. The DeadLockDetection module creates a graph of all the current locks and requests held by each thread in order to determine if a deadlock will occur. When a deadlock situation arises, an exception is thrown and passed through the ProtectedObject for the application to deal with.

The **Thread Viewer** is an optional stand-alone program. It serves as a debugging tool for the programmer. The programmer can view the schedules produced by the ConcurrencyControl. Corrupted data is highlighted in red to show where an exception would be thrown.

In order for many threads to safely use the same set of ProtectedObjects at the same time, a method of accessing data must be used. This is the job of the **ConcurrencyControl** module. The ConcurrencyControl implements a set of rules that allow threads to safely run at the same time. Obviously there are many different ways to allow the threads to interact with each other. The DynAC Model allows various implementations of ConcurrencyControl to be implemented with little difficulty.

These modules work together in the following manner: the Application makes requests (read, write or update) to a ProtectedObject before the data in ProtectedObject can be used. ProtectedObject passes the requests to the assigned ConcurrencyControl, which decides when the threads can access data. Because the DynAC Model does not demand that the ConcurrencyControl always return correct schedules, the ProtectedObject checks for the possibility of corrupted data and uses the DeadLockDetection module to check for deadlock. If no problem is found, the Application is allowed to proceed and use the data in ProtectedObject. If an error occurs, an exception is thrown and it is the responsibility of the Application to correct the problem. Meanwhile, the Thread Viewer collects the state of the system and graphically displays it.

## Implementation of the DynAC Model

### Application

The Application is a parallel program written using simple guidelines. Because most programmers have been only trained to think serially, making safe, parallel programs can be a difficult and dangerous process for them. Our model would make safe, parallel programming available to proficient serial programmers with only minimal knowledge of Java threads.

The Application is written in a serial manner with some easy rules for creating a safe, parallel environment. The first rule is that all shared data must be in logically separate pieces. These pieces are contained in or are instances of classes which extend ProtectedObject. The programmer could then choose an appropriate ConcurrencyControl for the Application (possibly from a library) to initialize the ProtectedObject. If the chosen ConcurrencyControl has a potential deadlock situation, the programmer will pass a DeadLockDetection instance to the Application's ProtectedObjects.

In order to make a Java parallel program, threads must concurrently run separate operations on shared resources. Whenever a thread needs data that is stored in a ProtectedObject, the thread asks the ProtectedObject if it can safely use it, using four methods: `read_Object`, `write_Object`, `update_Object` and `done`. Threads performing read, write and update operations call the corresponding methods. When a thread is done with that ProtectedObject, it simply calls `done` to release it.

If the ProtectedObjects are in a state where the data will become corrupted or deadlock will result, an exception of the proper type will be thrown. These exceptions must be caught by the programmer, who must deal with them in the Application. Each program is different. It is common to catch CorruptedDataExceptions and merely reread the ProtectedObject. This would allow operations to continue by giving the Application the correct version of the ProtectedObject to write to. When DeadLockExceptions are caught, all of the ProtectedObjects that the thread has permission to use must be released, using the `done` method. This would allow other threads waiting on any of those ProtectedObjects to access them, thus avoiding deadlock. In either case, the data is protected. The following code method demonstrates how to use the DynAC Model.

```
public boolean Check(ProtectedObject r1,r2)
{
  boolean ans;
  try
   { r1.read_Object();
     r2.read_Object(); }
   catch( DeadLockException e)
      { dropAllObjects();
         // All Protected Objects must be dropped
```

```
          restartThread(); }
     // Permission has been granted
     // Now we can read the Protected Objects
     if(r1.getKey() r2.getKey() )
     { try
       { r2.writeObject() }
       catch (DeadLockException e)
       { dropAllObjects();
         restartThread(); }
       catch (CorruptedDataException e)
       { return( Check(r1,r2) ); }
       r2.incrementValue();
       ans = true; }
     else
     { ans = false; }
   r1.done();
   r2.done();
   return(ans);
 }
```

## The ProtectedObject class

The ProtectedObject class is a class which extends Java's basic Object class. The ProtectedObject class creates a data object which can be shared between concurrent processes. It does this by adding specific variables and methods which enable the DynAC Model to keep track of the correct version of a data object, thus detecting corrupted data when it occurs.

The ProtectedObject class keeps track of the correct version by using a version-count algorithm, as specified below. Each ProtectedObject has a variable for the version and a hashtable, which stores the versions that each thread is requesting. If the operation is a read, then the entry in the hashtable is set to the version of the ProtectedObject. If the operation is a write and the version requested by the thread is not the current version, an exception is thrown. If the versions match, the ProtectedObject is written to and the version is incremented.

```
if operation = read
        VC[thread] = version;
else if operation = write
        if VC[thread] <> version
           throw CorruptedDataException;
        else
           version++;
```

The ProtectedObject class implements the version count algorithm by using four synchronized methods: `read_Object, write_Object, update_Object` and `done`. These methods work with the ConcurrencyControl class to determine when the method may be completed without causing data corruption. They also work with the DeadLockDetection class to check for possible deadlock situations. `read_Object` determines if the ProtectedObject is safe to be read, by checking the current version. `write_Object` determines if the ProtectedObject is safe to be written to, by checking the current version. It will only allow a write operation if the version requested matches the current version. When it is finished, it will increment the version for the ProtectedObject. `update_Object` determines if the ProtectedObject is safe to be updated, implementing the version count algorithm. Like `write_Object`, the versions must match for an update to be completed. Upon completion, `update_Object` will also increment the version for the ProtectedObject. `done` is called in order to release the lock placed on a ProtectedObject by a `read_Object, write_Object` or `update_Object` operation.

As an example of how the version-count algorithm and the ProtectedObject methods work, consider an ATM scenario in which Bob is depositing $100 and Jane is requesting the current balance on their account. If the starting balance is $500, the transactions could take place in the following order:

```
Bob starts his transaction on version 1 of the balance (the ProtectedObject)
Jane starts her transaction on version 1 of the balance (the ProtectedObject)
Bob deposits $100 (writes to ProtectedObject)
        and changes the balance to $600 and the version to 2
Jane tries to change the current balance (reads ProtectedObject)
        on version 1 and cannot do so because it is not the correct
        version.
```

In our DynAC model, the system would throw a CorruptedDataException here for the programmer to handle. However, in an actual banking system, if this occurred, Jane would probably never know. The system would recognize the new version and give her the new balance (perform a read_Object on ProtectedObject, version 2). The version-count algorithm enables the tracking of the current version of a ProtectedObject and the detection of data corruption by outside methods.

## The DeadLockDetection Class

Deadlock is an event where threads hold resources demanded by other threads, resulting in cross dependencies. This leads to a situation where threads wait on each other and suspend actions forever. The DeadLockDetection class is meant to serve as a warning system to detect when these cyclical dependencies occur.

The current module to detect deadlock uses two assumptions about the ConcurrencyControl. The first assumption is that a thread can only be in the act of requesting one ProtectedObject at any time. The threads are not allowed to try to get permission for a ProtectedObject while they are waiting for the use of another ProtectedObject. The current implementation of ProtectedObject ensures this criterion.

The second assumption depends on the ConcurrencyControl class' behavior. When the ConcurrencyControl receives a request, it views the system's current state and returns a boolean depending on whether the thread can continue in that state. For a deadlock to be detected, ConcurrencyControl must turn down thread **A** only if thread **B** is using the resources thread **A** needs. In other words, a thread that is denied the use of a resource is waiting for the resource to be released by another thread. However, there are times when more than one thread may be able to use the same ProtectedObject at the same time. This class is prepared to handle these multiple lock types.

DeadLockDetection uses three class methods to find out about the present state of the system. Each of these methods sends a ProtectedObject as a variable parameter. `Addlock` is called when the ConcurrencyControl allows the thread to use the ProtectedObject. `Removelock` is called when the thread notifies the ConcurrencyControl that it is done using the ProtectedObject. And finally, `Addrequest` is called when the ConcurrencyControl does not allow a thread to use a Protected Object. At this point the dependencies are checked. If a cyclic pattern is detected, an exception is thrown. Otherwise the request is stored and the system can continue.

Deadlock is detected by creating a graph of the dependencies and testing for cyclic behavior for each new request. The graph is implemented with 2 hashtables. The first hashtable, **ThreadTable**, is keyed to the currently running thread and maps to a class called ThreadRequest. **ThreadRequest** contains, in addition to the thread that is running, the ProtectedObject that the thread is requesting.

The second hashtable, **ObjectTable**, uses a ProtectedObject as a key and maps to a **Vector** containing ThreadRequests. These ThreadRequests contain threads which have permission to use this particular ProtectedObject. These structures are illustrated in Figure 2.

---

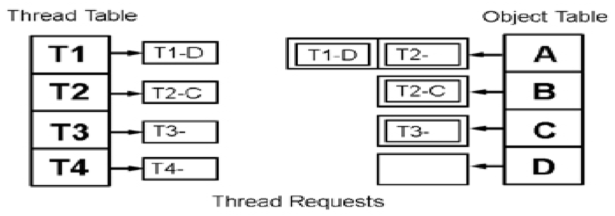
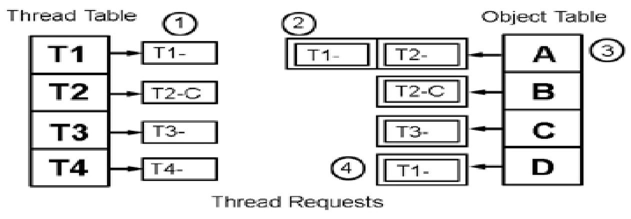


Figure 2: Deadlock Detection Implementation

---

When `Addlock` is called, the current thread is used to access the ThreadTable. The returned ThreadRequest has its ProtectedObject field set to null and saved in a temporary variable. Since Java objects are thinly disguised pointers, there exists only one ThreadRequest for each thread. Furthermore, when any Object is updated (such as ThreadRequests or Vectors) all instances of that Object will be updated. The ProtectedObject, which the thread has received permission to use, is used as the key in the ObjectTable. Then, the ThreadRequest is placed into the vector that is returned from the ObjectTable. `Removelock` removes the appropriate ThreadRequest from the vector that is returned from the ObjectTable. Figure 3 shows `Addlock` being called to grant ProtectedObject **d** to thread T1.

---



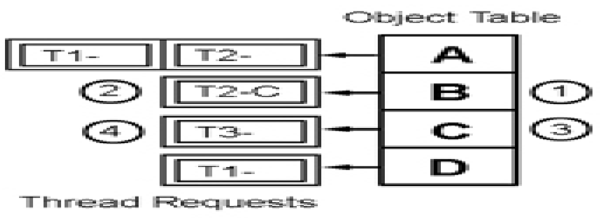
1. ThreadTable is updated with new ThreadRequest.
2. All ThreadRequests are updated simultaneously.
3. Use the ProtectedObject as the key in ObjectTable.
4. New ThreadRequest is placed in the Vector.

Figure 3: Add Lock Example

`AddRequest` calls a recursive algorithm shown below to check for a cycle.

```
checkRequest(PObject obj) throws DeadLockException
{
  Vector vec = (Vector) objectTable.get(obj);
  for(int i=0;i < vec.size();i++)
  {
    ThreadRequest req = (ThreadRequest) vec.elementAt(i);
    if(req's thread matches the current thread)
      throw new DeadLockException(``This Request will cause a Deadlock'');
    else if(req's ProtectedObject is not null)
      checkRequest(req's ProtectedObject);
  }
}
```

If a cycle is detected, an exception is thrown for the programmer to deal with. Otherwise, the ThreadTable is called with the current thread and then returns a ThreadRequest. The ThreadRequest's ProtectedObject field is updated to the Object that is currently being requested. Figure 4 shows an AddRequest being called to show a request of ProtectedObject **b** by thread T3. Since this will cause a cross dependency, a DeadLockException is thrown.



1. Use the Protected Object as a key in the table.
2. Check each ThreadRequest in the vector.
3. Since the Thread in the ThreadRequest does not match T3, we do not throw an exception and use ProtectedObject as a key in ObjectTable.
4. The thread in the ThreadRequest is equal to T3. A DeadLockException is thrown.

Figure 4: Add Request Example

## The ConcurrencyControl Class

In order for a system to effectively use several threads on shared resources, a Concurrency Control is necessary. It is the ConcurrencyControl class' job to tell Threads when they may proceed with their operations on the shared resources. In the DynAC Model, the ConcurrencyControl is never addressed directly by the Application. The system makes all requests for shared resources to the ProtectedObject, which protects the resources. In turn, the

ProtectedObject relays the requests to the ConcurrencyControl. The ConcurrencyControl decides if the action should be allowed and returns the permission. The final job of the ConcurrencyControl is to optionally deliver information to the Thread Viewer.

Since there are many different ways to implement a means of concurrently controlling shared resources, the DynAC Model implements the ConcurrencyControl as an abstract class. Each implementation of a new ConcurrencyControl requires a programmer to subclass the ConcurrencyControl and rewrite the methods. This allows different implementations to be used without affecting the other portions of the system. For example in the ATM problem, a simple system could be created that executes correctly but slowly. In order to create a faster form of ConcurrencyControl, only an instance of a different ConcurrencyControl is needed to change how the system executes. Several applications could use the same form of ConcurrencyControl. Therefore, ConcurrencyControls could be stored in libraries so that the programmer doesn't need to actually write the ConcurrencyControl.

Each subclass of ConcurrencyControl has a system of recording and viewing the current state. This could take the form of a table of values, a queue, or nothing at all, depending on the programmer's implementation. In addition, the abstract ConcurrencyControl class has four abstract methods that must be overwritten in any subclass: `OktoRead, OktoWrite, OktoUpdate`, and `release`. `OktoRead, OktoWrite,` and `OktoUpdate` all return booleans based on the current state. Each of these methods is called by a ProtectedObject to request permission for a Thread to use the ProtectedObject. The ConcurrencyControl checks the state of the Application to determine if the operation requested (read, write or update) should be granted. The method would then return a boolean as an answer. The final method that must be overwritten is `release`. This serves as a method for the system to let the ConcurrencyControl know when a thread is done with a ProtectedObject. All of the methods to communicate with the Thread Viewer are located in the superclass. In order to send information to the Thread Viewer, the method `writeMonitor` is used in the subclass.

## Viewing Resource Protection

In order to view resource protection, a ThreadViewer class was created to provide a visual aid in understanding the object and thread dependencies between processes. It is primarily intended as a debugging and analysis tool. A user should be able to see from the Viewer where the parallel processes cause data to be corrupted or a deadlock situation. We found this kind of visual tool very helpful when dealing with a multi-threaded program. Without a visual aid, it is often extremely difficult to truly understand what the program is doing or what problems are arising and where.

The Viewer has two diagrams to help the programmer in debugging his parallel program: Figure 5a shows the **Object Dependency Graph**. This diagram shows which thread operations are acting on which objects. It presents a clear picture, per data object, of the order in which operations are being performed by the threads. The second diagram, the **Thread Step Graph**, is shown in Figure 5b. This diagram is very useful because it can show the exact location at which data is corrupted. The Thread Step Graph shows, for each thread, the operations performed and data objects used. The arrows represent dependencies between the thread steps, i.e. thread step 2 cannot run until thread step 1 is completed. The version number for each data object is printed at the top left corner of each node, i.e. ``V0". Figure 5b also shows how corrupted data is represented in the Viewer. The node at which the data becomes corrupted is noted in red. This is accomplished using the version count algorithm, outlined earlier, and should be a very helpful aid for the programmer of a concurrent system. The Thread Step Graph can also show the programmer where deadlock has occurred in a system, by using the option to view the requests for locks. This shows which threads have requested which data objects and are waiting for locks. We have not shown this here since showing deadlock requires an extremely large graph.

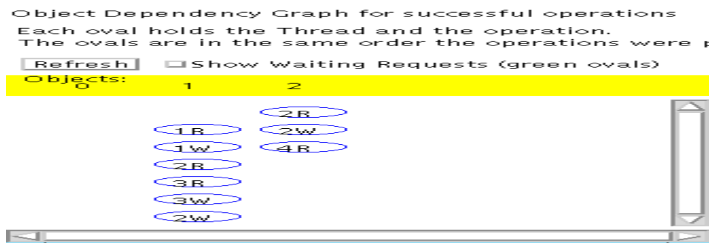

Object Dependency Graph for successful operations
Each oval holds the Thread and the operation.
The ovals are in the same order the operations were r

Figure 5a: Object Dependency Graph

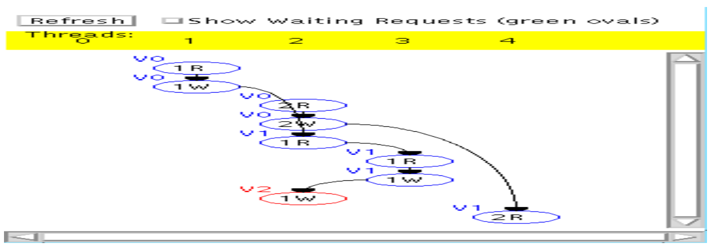


Figure 5b: Thread Step Graph without the requests

## Performance Concerns

There are several performance concerns which need to be addressed. The first concern is that using the DeadLockDetection class causes the application to run much slower than without the DeadLockDetection. Another concern is how the DynAC Model performs compared to a system which does not protect its data objects. We have taken three possible scenarios of the Dining Philosophers example in order to test performance. The first scenario does not use the ProtectedObject class, but rather uses an implementation of the Dining Philosophers from Tanenbaum [5]. The second scenario uses the DynAC Model without Deadlock Detection and the third scenario uses the DynAC Model with Deadlock Detection. Figure 6 illustrates the real time differences in the three scenarios. Each scenario was tested for 5, 10, 20, 50 and 100 philosophers, where each philosopher eats 5000 times. All other conditions for the tests were kept the same.

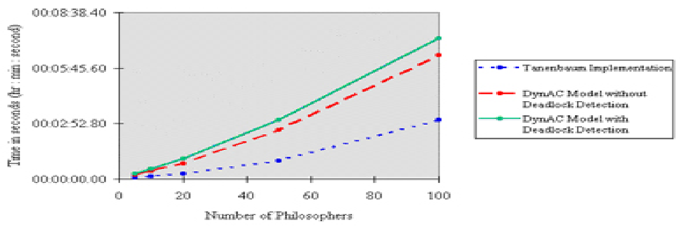


Figure 6: Real Time Comparison for Dining Philosophers Performance Tests
( Each philosopher eats 5000 times )

As you can see by the graph, the three scenarios each performed linearly in comparison with each other. This linear performance could change to a difference of up to n $^2$ if a Concurrency Control with multiple locks is used. The graph illustrates that using the DynAC model creates a significant performance penalty which increases when deadlock detection is added. The additional time for the deadlock detection results from the algorithm involved in the DeadlockDetection class. Every check for deadlock involves traversing a large, multi-arrayed graph to ensure that deadlock is not occurring. While the performance hit in using the DynAC Model is large, using the model and its classes ensures that the data objects will not be corrupted and that deadlock will be detected. The Tannenbaum implementation, or any other textbook implementation that we found for the Dining Philosophers, avoids deadlock and data corruption by using a specific knowledge of the system; however, it takes much more work to implement and to prove correctness. We recognize the performance degradation involved in using the DynAC Model and hope to reduce this to an acceptable level in the future, thus making it even more advantageous for a parallel program to use the DynAC Model.

## Conclusion

In this paper, we have shown the current manner in which parallel programs must be written in Java, making use of the `wait` and `notify` methods and making key data sharing methods synchronized. We discussed the need for shared resource protection and deadlock detection in Java and the lack of provisions for these key issues in the current Java implementation. We introduced the DynAC Model and elaborated upon each of its sections: the Application, the ProtectedObject, the DeadLockDetection, the ConcurrencyControl and the Thread Viewer. We illustrated how a programmer could use these classes to write parallel programs, using the Dining Philosophers and ATM examples. Finally, we analyzed the results of performance tests using the Dining Philosophers example.

There are several directions to take with our DynAC Model for parallel programming. We hope to correct the performance decline that we observed when the Model was used, in addition to making Deadlock Detection less time consuming. We would like to better enable the programmer to use the Thread Viewer for large programs with large numbers of shared data objects. Ultimately, we would like to see Java include more reliable ways for protecting shared resources and detecting deadlock.

## References

**1**

Dijkstra, E.W. Co-operating Sequential Processes. *Programming Languages*, Genuys, F. (ed.), Academic Press, 1965.

**2**

Hoare, C.A.R. Monitors, An Operating System Structuring Concept. *Communications of the ACM* , 17, 10 (Oct. 1974), pp. 549-557.

**3**

Sealevel Software, *The SWMRG Class and Test Applet* . http://www.halcyon.com/sealevel/testswmrg.htm , 1997.

**4**

Sun Microsystems, *JDK 1.1 Documentation* , http://java.sun.com/products/jdk/1.1/docs/api/packages.html, 1997.

**5**

Tanenbaum, A.S., *Operating Systems: Design and Implementation* . Prentice-Hall, 1987.

## Acknowledgments

We would like to acknowledge the 1997 National Science Foundation REU program under which we were able to do the research which led to this paper. We would also like to acknowledge Hope College and its Computer Science department for their role in our work. Finally, we would like to acknowledge our research mentor, Mike Jipping, Associate Professor of Computer Science at Hope College, for his guidance, assistance and the inspiration behind this research project.

## Author Biographies

Anita Van Engen is currently a Senior at Hope College in Holland, Michigan and is from southern California. She will graduate in May 1998 with a Computer Science major and double minors in Math and Spanish. Her interests include Java, computer graphics, volleyball and spending time with her friends and fiance. She is a member of Hope College's Alcor Chapter of MortarBoard, ACM and is the 1997 General Chair of Nykerk, a traditional freshman-sophomore competition at Hope.

Michael K. Bradshaw is currently a Junior at Centre College in Danville, KY. He is a double major in Math and Computer Science. While maintaining an interest in physical sciences, Michael is also an officer for the local APO chapter. He is currently attempting to become a Linux guru, a programming whiz, a Grad Student and a teacher to all.

Nathan Oostendorp is a native of Zeeland, Michigan and attends Hope College in Holland, Michigan where he is currently a Junior. He is a combined English/Computer Science major and in addition to doing NSF research also works in the R&D department of the Donnelly Corporation where he programs Vision and Data Aquisition Systems. He is an officer in the local chapter of the ACM, an avid Linux fan, and also enjoys writing silly Perl scripts.