Deadlock Detection Using DTrace

CS 671 – Project I

OS Observability Tools

Douglas S. Corner
G00081155
March 27, 2008

1. Summary

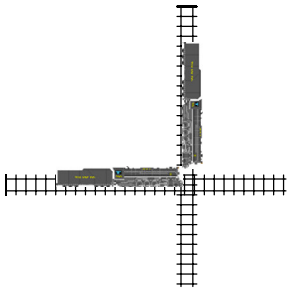
Recently low cost multicore processors have become readily available. In order to exploit this capability it is generally necessary to add multi-threading to exiting applications. Programmers need to understand the timing variability that now becomes an issue and to be aware of the possibility of getting into deadlock situations, common in multi-threaded code. The problem to be solved in this project was to detect deadlock in a java application using the dtrace hotspot provider. This also provided an introduction to the monitoring of application code.

The application chosen to study was the well known Dining Philosophers problem first described by Edgar Dijkstra in 1965. A java implementation of Dining Philosophers was written for this project. No instrumentation was added to the program other than making the program quite modular so that method entries and returns could be traced. As it turns out, getting the program to deadlock isn't very predictable and the wait was sometimes long.

A dtrace script was written to track requests to obtain and release chopsticks which were implemented as binary semaphores. The Philosophers were implemented as separate thresds. Some surprising problems were encountered while doing the tracing in that just observing method names could cause dtrace errors. In addition current limitations of the D programming language became quite obvious.

The dtrace script was developed and was successfully able to detect deadlocked conditions in the Dining Philosophers program.

2. Overview of Deadlocks



It is said that a law of the state of Kansas states: “When two trains approach each other at a crossing, both shall come to a full stop and neither shall start up again until the other has gone. While this may not really be a law it is illustrative of a common operating system problem. More properly we define a deadlock as the situation where two or more processes are waiting for an event that can be caused only by one of the waiting processes. [Silbershatz p. 204]. Deadlocks occur when we try to manage the allocation of dynamically used resources. These are objects that go through the following sequence:

- Request – A process makes a request for a resource and waits if it isn’t available.
- Use – The process will need to have use of all required resources before its work can be completed.
- Release – The process releases resources once they are no longer needed so that other processes can use them.

For example, if we have the following where P1 and P2 are processes and A and B are binary semaphores initialized to 1:

Step	P_1	P_2
1	wait(A);	
2	preemptive context switch	
3		wait(B);
4		preemtive context switch
5	wait(B);	
6	context switch - waiting for B	
7		wait(A);
8		context switch - waiting for A

Following step 8, P_1 is holding A and waiting for B. P_2 is holding B and waiting for A. This system is now deadlocked and cannot proceed.

Deadlocks can occur if four necessary conditions hold simultaneously:

- Mutual exclusion: Only one of the involved processes can use a resource instance at a time.

- Hold and wait: A process holding at least one resource is waiting to acquire additional resources held by other processes.

- No preemption: A resource can only be released voluntarily by the process holding it, and only after that process has completed its task.

- Circular wait: There exists a set $\{P_0, P_1, \ldots, Pn\}$ of waiting processes such that P_0 is waiting for a resource that is held by P_1, P_1 is waiting for a resource that is held by P_2, $\ldots, P_{n-1}$ is waiting for a resource that is held by P_n, and P_n is waiting for a resource that is held by P_0.

These are known as the *Coffman Conditions* [Coffman 1971]. Coffman showed that deadlock can only occur in systems where all four conditions happen.

Examples of situations where deadlock can occur are:

- A host system with one tape drive and one printer. Program A requests the tape drive and then the printer. Program B requests the printer and then the tape drive.

- A database system. Transaction 1 locks the customer record and then tries to lock a product inventory record. Transaction 2 locks the inventory record and then the customer record.

3. Deadlock Detection

The heart of the problem is that deadlocked processes don't know they are deadlocked. Much of the early work on deadlock characterization is due to Dijkstra [Dijkstra 1971].

Deadlock problems are commonly described by a system resource-allocation graph first described by R.C. Holt [Holt 1972]. The following description of this process is taken from [Silberschatz].

The resource-allocation graph is a directed graph consisting of a set of vertices and edges. The vertices consist of processes (P_x) and resources (R_y). The edges either represent requests, $P_i \rightarrow R_j$ or assignments $R_k \rightarrow P_l$. The request $P_i \rightarrow R_j$ indicates that process i has requested and is waiting for an instance of resource j. The assignment $R_k \rightarrow P_l$ is an indication that an instance of resource k has been assigned to process l. Figure I consists of three processes P_1, P_2, P_3. and four different types of resources, one instance each of R_1 and R_3, two instances of R_2 and three instances of R_4. Process P_1 holds one instance of R_2 and is waiting for R_1.

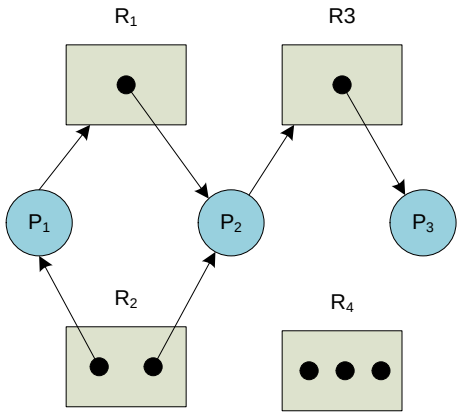


Figure I
Resource-allocation graph
No deadlock

Holt showed that if the graph contains no cycles, as in Figure I, then no process in the system is deadlocked. He also showed that if cycles do exist then a deadlock is possible. A cycle is then a necessary but not sufficient condition for a deadlock.

In figure II two cycles exist:

$$R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_2 \rightarrow R_3$$

and

$$R_1 \rightarrow P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_1 \rightarrow R_1$$

Figure II describes a condition where all three processes P_1, P2, and P3 are deadlocked.

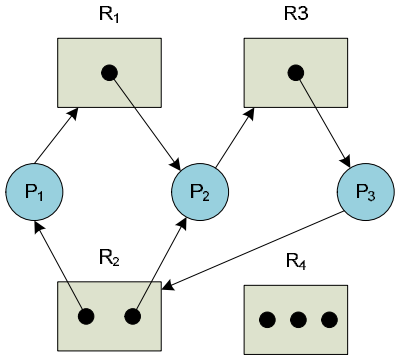


Figure II
Resource-allocation graph
With a deadlock

If there are multiple instances of resources then it becomes possible to have a cycle but no deadlock. In figure III a cycle exists $R_1 \rightarrow P_3 \rightarrow R_2 \rightarrow P_1 \rightarrow R_1$. P_1 and P_3 are suspended waiting for resources, however, as soon as P_4 releases its copy of R_2 the temporary cycle will be broken and the program can proceed.

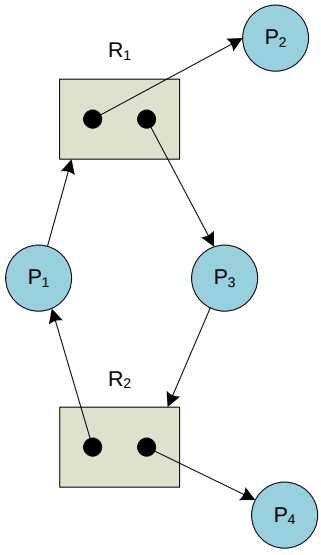


Figure III
Resource-allocation graph
Cycle but no deadlock

4. Deadlock Prevention

There are a number of algorithms available for preventing or avoiding deadlocks, two of which are discussed here.

In order to create a deadlock it is necessary to fulfill all four of the Coffman conditions. Eliminating any one of the conditions will ensure that deadlock cannot occur [Havender 1968].

- Mutual exclusion: The use of sharable resources will prevent deadlocks. Examples are read only files and the use of print spoolers rather than dedicated printer access. In general there will be times when exclusive access to a resource is essential.

- Hold and wait: If a process is not allowed to hold one resource while waiting for another then this condition would not take place. One approach is to request all resources at once. This may be possible but will often be found to cause an inefficient use of resources.

- No preemption: It may be possible in some cases to preempt a process and take a resource from it that is needed by another one. Virtual memory is one example of this type of operation in that memory can be taken from one process, given to another and then later restored.

- Circular wait: Creating a total ordering of all allocateable resources will eliminate this condition. Resources are allocated in accordance with this ordering. Processes must then allocate resources in order thus eliminating circular waiting.

While any of the above actions will eliminate the chance of deadlocks they all impose restrictions on applications. and generally impractical.

The second technique is known as the Bankers Algorithm [Dijkstra Undated]. It is assumed that there are multiple instances of each resource type. This implies that a resource-allocation graph can not be used. Processes are required to declare in advance the maximum need for each resource. Over time, processes make actual requests which cannot exceed the predeclared maximum quantity. If the allocation leaves the system in a “safe” state then the system grants the resource. If the system would be left in an “unsafe” state then the process is suspended until the system has adequate resources to grant the request. There is an algorithm defined that determines if a particular allocation would leave the system in an “unsafe” condition and therefore in danger of being deadlocked.

Silberschatz [Silberschatz] succinctly sums up the three approaches used in the real world.

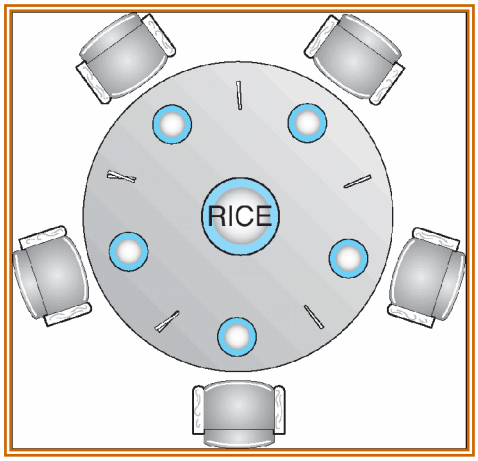
- Use some protocol to prevent deadlocks.

- Allow the system to enter a deadlock state, detect it and recover

- Ignore the problem altogether.

The third approach is the one used on modern operating system.

5. The Dining Philosophers Problem



The Dining Philosophers problem has been used for years to study deadlock and process coordination problems. [Dijkstra 1965a] and [Dijkstra 1971]. The problem is stated as follows:

- Five philosophers are seated around a round table. There are a total of five chopsticks.
- Each philosopher eats for a while and then thinks for a while.
- In order to eat, the philosopher must pick up two chop sticks. Note that the left chopstick for one philosopher is the right chopstick for his neighbor to the left.
- In the base problem the philosopher picks them up one at a time and waits if it isn't available.

The problem, of course, is that the system may reach the condition where each philosopher has picked up his left chopstick and is waiting for his neighbor to the right to put down his left chopstick. This is a classic deadlock condition.

This problem was coded in java. Each philosopher is implemented as a separate thread. Each chopstick is implemented as a binary Semaphore. Each philosopher goes through the following steps

Pick up left chopstick
Pick up right chopstick
Eat for a random time
Put down left chopstick
Put down right chopstick
Think for a random time
Repeat

The java program that implements this is included as an appendix to this report.

The goal of this project is to detect that deadlock has occurred without human assistance.

6. Description of Solution

The basic detection technique is to determine that a loop exists; that each process is waiting for a resource held by another process which in turn is waiting for a resource held by another process and that this ultimately constitutes a loop. This is a fairly simple system in that each process only requests two resources and that it does so in sequence, first the left chopstick and then the right one. A deadlock is visually detected by noting that all processes are waiting for their right chopsticks. The problem is to instrument a monitor that detects the state of the java program and that detect that all processes are locked.

The dtrace script does this by capturing all method-entry and method-return invocations. When a thread enters either pickUpLeft() or pickUpRight() it increments a variable called *waiting*. When the thread exits pickUpLeft() or pickUpRight() it decrements the *waiting* varable.

Within the pickUpX routines the thread will execute

```
chopsticks[id].acquire();
```

The chopsticks variable is an array of java Semaphores each initialized with one permit. If the chopstick is in use by this philosophers neighbor then the thread will block until the neighbor enters putDownLeft() or putDownRight() and executes:

```
chopsticks[id].release();
```

At that point the blocked thread will resume and will exit the pickUpX() method. The dtrace monitor-return probe will fire and the *waiting* variable will be decremented. The result is that at all time the dtrace *waiting* variable reflects the number of threads that have entered pickUpX() methods but have not yet returned.

It is sometimes possible that all five threads are executing in a pickUpX method simultaneously but that the system is not yet deadlocked. This is a transient condition due to preemptive scheduling by the operating system. To guard against false alarms the *waiting* variable is examined in a profile-1 probe which fires once per second. A second variable, *seq,* is used to ensure that deadlock will only be detected if *waiting* is equal to five for three consecutive seconds.

Script used to run the application and the dtrace script:

```
# Start DP program in a separate window
gnome-terminal -e 'java -XX:+ExtendedDTraceProbes -jar dist/dp.jar'&

# We need a small delay so that the pid is available for dtrace
sleep 1

# Start dtrace.  Pass the pid of the DP program as $1
./dp1.d `pgrep -f dp.jar`
```

The dtrace script developed to detect deadlock in the Dining Philosophers program is given below:

```
#!/usr/sbin/dtrace  -qs
#pragma D option dynvarsize=64m

/*
    Douglas Corner - G00081155
    George Mason University
    CS 671 - Project I - Observability Tools

    DTrace Script to detect deadlocked in a multithreaded program.
    We keep track of the number of waiting threads by trapping certain Java method calls.
    As a thread enters a PickUp (left or right) method we consider that thread as waiting
    by incrementing waiting.  When the thread returns from the method the waiting variable is decremented
    If the thread is forced to wait to get the choppstick (Semaphore) the waiting variable will stay
    incremented.

    By using a profile monitor which executes once per second we watch the number of waiting threads.  If the
    number stays at 5 (the number of philosophers) for three seconds, this indicates that all threads are waiting and
    the program is deadlocked.
*/

/* Start of trace   */
dtrace:::BEGIN
{
    printf("Starting - pid = %d\n", $1);            /* $1 is pid of Dining Philosopher java program */
    startTime = timestamp;                          /* Time started */
    waiting = 0;                                    /* Number of threads currently waiting */
    seq = 0;
}

/*
Only examine methods from the "Philosopher" class. Even trying to look at method names from other classes
causes DTrace errors
*/
```

```
/* pickUpLeft entry */
hotspot$1:::method-entry
/strstr(copyinstr(arg1), "Philosopher")!= NULL && copyinstr(arg3) == "pickUpLeft"/
{
    ++waiting;
    /*printf("Pick Up Left entry - Waiting=%d\n", waiting);*/
}

/* pickUpLeft return */
hotspot$1:::method-return
/strstr(copyinstr(arg1), "Philosopher")!= NULL && copyinstr(arg3) == "pickUpLeft"/
{
    --waiting;
    /*printf("Pick Up Left return - Waiting=%d\n", waiting);*/
}

/* pickUpRight entry */
hotspot$1:::method-entry
/strstr(copyinstr(arg1), "Philosopher")!= NULL && copyinstr(arg3) == "pickUpRight"/
{
    ++waiting;
    /*printf("Pick Up Right entry - Waiting=%d\n", waiting);*/
}

/* pickUpRight return */
hotspot$1:::method-return
/strstr(copyinstr(arg1), "Philosopher")!= NULL && copyinstr(arg3) == "pickUpRight"/
{
    --waiting;
    /*printf("Pick Up Right return - Waiting=%d\n", waiting);*/
}


/*Once per second display the number of waiting and number of times we found waiting equal to 5 */

profile-1
{
    printf("Waiting=%d   seq=%d\n", waiting, seq);
}


/*

Once per second see if waiting had been 5 and then dropped down.  This can be caused by transient conditions
due to thread preemption.  If this is true set seq back to 0
```

```
*/
profile-1
/waiting < 5 && seq > 0/
{
    seq = 0;
}


/* If waiting is 5 then increment seq */

profile-1
/waiting == 5/
{
    ++seq;
}

/*
    If seq is 3 then we had five waiting threads for three seconds and have a deadlock
    Display a message and exit.
*/

profile-1
/seq == 3/
{
    printf("Deadlock detected!!\n");
    exit(1);
}
```

7. Test Results and Discussion

Results

The following listing is status output from the Dining Philosophers java implementation. It runs continuously even after becoming deadlocked. A deadlocked condition is indicated by the fact that all five threads are waiting to pick up the right chopstick (State = RRRRR). As it took around 30 minutes (1789.724 seconds) for the deadlock to occur on this run, most of the status output has been deleted. The meaning of the columns on the output is as follows:

Time Seconds from start of run

Msg# Consecutive number of messages (Sometimes printed out of order)

P# Philosopher number (0..4)

W Current number of philosophers waiting for a chopstick

State An array of characters describing the current state of each philosopher:

- t - Thinking
- e - Eating
- p - Eating complete, putting down chopsticks
- L - Waiting for left chopstick
- l - Have left chopstick
- R - Waiting for right chopstick
- r - Have right chopstick
- \- - Just initialized or between cycles of eating and thinking

Time	Msg #	P#	W	State	Action
1787.981	41262	1	3	LlRRe	Picked up left
1788.023	41263	4	3	LlRRp	Putting down left
1788.064	41264	1	4	LRRRp	Waiting for right
1788.104	41265	3	3	LRRrp	Picked up right
1788.146	41266	3	3	LRRep	Eating
1788.190	41267	3	3	LRRpp	Putting down left
1788.232	41268	4	3	LRRpp	Putting down right
1788.275	41269	2	2	LRrpp	Picked up right
1788.317	41270	2	1	lRept	Eating
1788.359	41271	3	1	lRept	Putting down right
1788.402	41272	2	1	lRppt	Putting down left
1788.444	41273	0	1	lRptt	Picked up left
1788.488	41274	4	0	lrptt	Thinking
1788.531	41275	0	1	Rrptt	Waiting for right
1788.574	41276	2	1	Rrpt-	Putting down right
1788.613	41277	1	1	Rrpt-	Picked up right
1788.655	41278	1	1	Rett-	Eating
1788.697	41279	3	1	Rett-	Thinking
1788.738	41280	1	1	Rptt-	Putting down left
1788.780	41281	2	1	Rpt--	Thinking
1788.821	41282	4	0	rpt--	Picking up left
1788.862	41283	2	1	rpL--	Waiting for left
1788.905	41284	1	1	rpL-l	Putting down right
1788.948	41285	0	1	rpL-l	Picked up right
1788.990	41286	0	0	etl-l	Eating
1789.034	41287	0	0	ptl-l	Putting down left
1789.077	41288	3	0	ptl-l	Picking up left
1789.117	41289	0	0	ptl-l	Putting down right
1789.158	41290	2	0	ptlll	Picked up left
1789.202	41291	1	0	ttlll	Thinking

```
1789.243       41292   4    1  ttRll  Picked up left
1789.284       41293   1    1  t-Rll  Picking up left
1789.327       41294   2    1  t-Rll  Waiting for right
1789.369       41295   0    1  tlRll  Thinking
1789.412       41296   3    1  tlRll  Picked up left
1789.454       41297   3    2  -lRRl  Waiting for right
1789.497       41298   0    2  -lRRl  Picking up left
1789.540       41299   1    2  -lRRl  Picked up left
1789.580       41300   1    3  lRRRl  Waiting for right
1789.620       41301   4    3  lRRRl  Picking up right
1789.663       41302   0    3  lRRRl  Picked up left
1789.704       41303   0    4  RRRRl  Waiting for right
1789.724       41304   4    5  RRRRR  Waiting for right
```

At this point the system is deadlocked.

Output of DTrace Script

Once per second the dtrace script prints the number of waiting threads and the number of times (in a row) that the number waiting was found to be five.

```
Waiting=4   seq=0
Waiting=4   seq=0
Waiting=4   seq=0
Waiting=3   seq=0
Waiting=3   seq=0
Waiting=3   seq=0
Waiting=3   seq=0
Waiting=3   seq=0
Waiting=3   seq=0
Waiting=3   seq=0
Waiting=3   seq=0
Waiting=4   seq=0
Waiting=4   seq=0
Waiting=3   seq=0
Waiting=3   seq=0
Waiting=3   seq=0
Waiting=3   seq=0
Waiting=3   seq=0
Waiting=3   seq=0
Waiting=3   seq=0
Waiting=3   seq=0
Waiting=3   seq=0
Waiting=3   seq=0
Waiting=5   seq=0
Waiting=5   seq=1
Waiting=5   seq=2
Deadlock detected!!
```

Discussion

This project successfully detected deadlock without modifying the application. It is, however, quite difficult to build a general purpose deadlock detector. This was a simple program implementing a problem that has been well studied over the years and didn't require extensive digging into the application.

There are certainly other techniques that might have been used. Other providers would have given similar information about locking and unlocking semaphores. Using monitor-enter and monitor-return provided a straight forward way to solve this problem.

It may be possible to use techniques like detecting that the application has not used any CPU cycles for some period of time and assuming that it must be deadlocked. This is quite risky as there may be legitimate reasons why no cycles were used, e.g. waiting for user input.

DTrace can be difficult to use. There appear to be substantial issues with respect to paging that can cause problems. One version of the program was tested by piping the output to grep to look for certain entries. This caused memory access problems apparently because of added delay. It was found (After much difficulty and with the eventual help of Sun personnel) that there are times that just looking at a method name causes problems. It was necessary to insert logic that checked the class name first and then the method name in a properly sequences AND predicate:

```
/strstr(copyinstr(arg1), "Philosopher")!= NULL && copyinstr(arg3) == "pickUpLeft"/
```

The "D" language is a bit crude. The lack of if-then-else constructs and any sort of looping make many approaches to analyzing more complex issues. A more traditional graph approach to detecting deadlock would require a much more flexible programming language.

The hotspot provider appears to be largely aimed at debugging internal java problems rather than java applications. It may be that there are quite adequate debugging facilities in tools like NetBeans and Eclipse. Sometimes it is difficult to diagnose java application issues with a tool that is itself a java application. It would be helpful\ to have an external tool like dtrace to dig deeper into applications.

An attempt was made to try to get access to variables internal to the application. It may be true that there is a way to do this but it wasn't obvious what it is. For example, the hotspot:::method-entry probe passes a pointer the method "Signature". Adding the following statement to the dtrace script:

```
printf("Signature: %s\n", copyinstr(arg5));
```

resulted in this output:

```
Signature: ()V
```

It isn't clear what this indicates but wasn't found to be helpful.

During development and as a result of performing some web searches a number of apparently undocumented features were uncovered, for example a number of string functions. This would lead one to assume that improvements to the language can be expected.

It took a surprisingly long time to get the application to deadlock. The displayed results do reflect a real deadlock. The nature of multi-threaded timing problems is obviously that they only fail when you don't want them to.

This project did provide some very useful experience that will prove useful in analyzing other java applications. As Solaris isn't considered by most researchers to be a traditional research platform, it would be very useful to have a dtrace implementation on Linux.

8. Bibliography

- [Coffman 1971] Coffman, E.G. et al, *System Deadlocks*, Computing Surveys, Vol. 3, No. 2, June 1971

- [Dijkstra 1965] E.W.Dijkstra. "Cooperating Sequential Processes" (EWD 123), Technical Report, Technological University, Eindhoven, the Netherlands (1965)

- [Dijkstra Undated] E.W. Dijkstra. "Een algorithme ter voorkoming van de dodelijke omarming" (EWD 108) Technical Report, Technological University, Eindhoven, the Netherlands (Undated)

- [Dijkstra 1977] E.W. Dijkstra. "EWD623: The mathematics behind the Banker's Algorithm" from *Selected Writings on Computing: A Personal Perspective*, Springer-Verlag, 1982. ISBN 0-387-90652-5.

- Dijkstra, E. W. (1971, June). Hierarchical ordering of sequential processes. Acta Informatica 1(2): 115-138.

- [Havender 1968] J.W. Havender, "Avoiding Deadlock in Multitasking Systems", IBM Systems Journal, Volume 7, Number 2 (1968), pages 74-84

- [Holt 1972] R.C. Holt, "Some Deadlock Properties of Computer Systems", *Computing Surveys*, Volume 4, Number 3 (1972), pages 179-196

- [Silberschatz] Silberschatz, Abraham; Peterson, James L. (1988). *Operating Systems Concepts*. Addison-Wesley. *ISBN 0-201-18760-4*.

9. Appendix

Java Source code for Dining Philosophers Implementation

```
/*
 * Main.java
 *
 * Created on February 29, 2008, 12:05 PM
 *
 * To change this template, choose Tools | Template Manager
 * and open the template in the editor.
 */
package dp;
import java.util.concurrent.Semaphore;
import java.lang.Thread;
import java.util.Random;

/**
 *  Program to implement the Dining Philosophers problem.
 *  George Mason University
 *  CS 671 - Project I - Observability Tools
 *  Douglas Corner - G00081155
 * @author dcorner
 */
```

```
/*********************
 * Class DinPhil      *
 *********************/

public class DinPhil {
    public static final int NUM_PHIL = 5;                          // Number of simultaneous philosophers
    public static final int THINK_TIME =  0;                       // Time between eating times
    public static final int EAT_TIME = 0;                          // Time spent eating
    public static final int INITIAL_SLEEP_TIME = 10000;            // Delay a while to wait for dtrace to start
    public static final int STATUS_UPDATE_TIME = 60000;
    public static final boolean DISPLAY_DETAIL_STATUS = true;      // Display status on every action
    public static long clock = 0;
    public static int msgNum = 0;
    public static int numWaiting = 0;
    // state is an array of characters describing the current state of each philosopher
    // t - Thinking
    // e - Eating
    // p - Eating complete, putting down chopsticks
    // l - Have left chopstick
    // L - Waiting for left chopstick
    // r - Have right chopstick
    // R - Waiting for right chopstick
    // - - Just initialized or between cycles of eating and thinking

    public static char[] state;

    // Chopsticks are implemented as an array of Semaphores each with a capacity of one.
    static Semaphore[] chopsticks;

    // Each philosopher is a separate thread implemented in the Philosopher class.
    static Philosopher[] p;

    Random rand = new Random();

    /*********************
     * DinPhil constructor*
     *********************/

    // Called from main()

    public DinPhil() {
        int i;
        System.out.printf("Program started.  Main ThreadID = %d\n", Thread.currentThread().getId());

        clock = System.currentTimeMillis();                    // Start time for run

        // Initialize state array
```

```
        state = new char[NUM_PHIL];
        for (i=0; i<NUM_PHIL; ++i)
            state[i] = '-';

        // Initialize chopsticks
        chopsticks = new Semaphore[NUM_PHIL];           // Create the array
        for (i=0; i< NUM_PHIL; ++i)
            chopsticks[i] = new Semaphore(1, true);     // Create each chopstick

        // Initialize Philosophers
        p = new Philosopher[NUM_PHIL];                  // Create the array
        for (i=0; i<NUM_PHIL; ++i) {
            p[i] = new Philosopher(i);                  // Create philosopher thread object
            p[i].start();                               // Start it running
        }   // for
    }
```

```
/*********************
 * DinPhil main()    *
 *********************/
public static void main(String[] args) {
    int i;

    // Sleep a while to let DTrace script start
    if (INITIAL_SLEEP_TIME > 0)
        try {Thread.sleep(INITIAL_SLEEP_TIME);} catch (InterruptedException ie) {}

    // Do initialization of main class
    DinPhil d = new DinPhil();

    System.out.println("All threads started\n");

    // Philosopher threads never return so we should wait forever to join p[0]
    try {
        for (i=0; i< NUM_PHIL; ++i)
            p[i].join();
    } catch (InterruptedException ie) {}
}   // main

 /*************
 * debug()    *
 **************/
 // Called by any of the Philosopher threads.  Could be used by main thread
synchronized static void debug(String msg, int id) {

    ++msgNum;
    String s = new String(state);           // Convery character array to string
    System.out.printf("%12.3f  %10d  %2d  %d  %s  %s\n",
            (System.currentTimeMillis() - clock)/ 1000.0, msgNum, id, numWaiting, s, msg);
}   // debug
```

```
/*********************
 * Class Philosopher *
 *********************/
// One object of this class will be instantiated for each philosopher

class Philosopher extends Thread {

    // Instance variables
    int id;                     // Local ID
    long ThreadID;              // ID assigned to thread
    int r;                      // Random number

    // Philosopher constructor
    Philosopher(int i) {
        id = i;
        ThreadID = this.getId();
        System.out.printf("Thread created.  Local ID=%d.  ThreadID = %d\n", id, ThreadID);
    }

/************
 * run ()   *
 ************/

    public void run() {
        try {
            // Random initial sleep time
            r = (int)(rand.nextDouble() * 20.0);
            Thread.sleep(r);

            // Main eat-think loop.  Executed continuously
            while (true) {
                state[id] = '-';
                pickUpLeft();           // Pick up left chopstick.  Wait if not available
                pickUpRight();          // Pick up right chopstick.  Wait if not available
                eat();                  // Sleep while eating
                putDownLeft();          // Put down left.  Will not wait
                putDownRight();         // Put down right.  Will not wait
                think();                // Sleep while thinkging
            }   // while

        } catch(InterruptedException e) {};
    }   // run
```

```
/******************
 * pickUpLeft ()   *
 ******************/

    synchronized void pickUpLeft() {
        try {
            // The two paths exist in order to display actual current status
            // It is possible that this thread will be prempted and status might be wrong.
            // This doesn't effect the logic of the program
            if (chopsticks[id].availablePermits() == 1) {           // See if chopstick is available
                if (DISPLAY_DETAIL_STATUS)
                    debug("Picking up left", id);
                chopsticks[id].acquire();                           // Pick up left chopstick.  Wait if not available
                state[id] = 'l';
                if (DISPLAY_DETAIL_STATUS)
                    debug("Picked up left", id);
            } else {
                state[id] = 'L';
                incrWaiting();
                if (DISPLAY_DETAIL_STATUS)
                    debug("Waiting for left", id);
                chopsticks[id].acquire();                           // Pick up left chopstick
                state[id] = 'l';
                decrWaiting();
                if (DISPLAY_DETAIL_STATUS)
                    debug("Picked up left", id);
            }
        } catch (InterruptedException ie) {}
    }   // pickUpLeft

/******************
 * pickUpRight()   *
 ******************/

    synchronized void pickUpRight() {
        // Same logic for if else statements as for pickUpLeft
        try {
            if (chopsticks[(id + 1) % NUM_PHIL].availablePermits() == 1) {  // See if available
                if (DISPLAY_DETAIL_STATUS)
                    debug("Picking up right", id);
                chopsticks[(id + 1) % NUM_PHIL].acquire();            // Pick up right chopstick
                state[id] = 'r';
                if (DISPLAY_DETAIL_STATUS)
                    debug("Picked up right", id);
            } else {
                state[id] = 'R';
```

```
                incrWaiting();
                if (DISPLAY_DETAIL_STATUS)
                    debug("Waiting for right", id);
                chopsticks[(id + 1) % NUM_PHIL].acquire();             // Pick up right chopstick
                state[id] = 'r';
                decrWaiting();
                if (DISPLAY_DETAIL_STATUS)
                    debug("Picked up right", id);
            }
        } catch (InterruptedException ie) {}
    }   // pickUpRight

/***********
 * eat()   *
 ***********/
    synchronized void eat() {
        try {
            state[id] = 'e';                                  // Status to eating
            if (DISPLAY_DETAIL_STATUS)
                debug("Eating", id);
            r = (int)(rand.nextDouble() * EAT_TIME);          // Sleep random amount of time
            Thread.sleep(r);                                  // Eat (sleep)
            state[id] = 'p';                                  // Eating done.  Put chopsticks down.
        } catch (InterruptedException ie) {}
    }

/******************
 * putDownLeft()   *
 ******************/

    synchronized void putDownLeft() {
        if (DISPLAY_DETAIL_STATUS)
            debug("Putting down left", id);
        chopsticks[id].release();                              // Put down left chopstick - No waiting
    }   // putDownLeft

 /******************
 * putDownRight()   *
 ******************/
   synchronized void putDownRight() {
        if (DISPLAY_DETAIL_STATUS)
            debug("Putting down right", id);
        chopsticks[(id + 1) % NUM_PHIL].release();             // Put down right chopstick - No waiting
    }   // putDownRight
```

```
    /******************
     * think()   *
     ******************/

        synchronized void think() {
            try {
                state[id] = 't';                                // Set state to thinking
                if (DISPLAY_DETAIL_STATUS)
                    debug("Thinking", id);
                r = (int)(rand.nextDouble() * THINK_TIME);      // Compute random number from 0 - THINK_TIME
                Thread.sleep(r);                                // Think (Sleep)
            } catch (InterruptedException ie) {}
        }   // think

    // Manipulate numWaiting in synchronized methods

    /******************
     * incrWaiting()   *
     ******************/

        synchronized void incrWaiting() {
            ++numWaiting;
        }   // incrWaiting

    /******************
     * decrWaiting()   *
     ******************/

        synchronized void decrWaiting() {
            --numWaiting;
        }   // decrWaiting

    }   // Class Philospher

}   // class DinPhil
```