A MUTUAL EXCLUSION ALGORITHM
BASED ON REQUEST AND FAIR ACCESS

BY
JINGSONG ZHANG
Bachelor of Science
Nankai University
Tian Jin, China
1992

Submitted to the Faculty of the
Graduate College of
Oklahoma State University
in partial fulfillment of
the requirements for
the Degree of
MASTER OF SCIENCE
December 2006
A MUTUAL EXCLUSION ALGORITHM
BASED ON REQUEST AND FAIR ACCESS

Thesis Approved:

Dr. M. H. Samadzadeh

______________________________________________
Thesis Advisor

Dr. G. E. Hedrick

______________________________________________

Dr. J. P. Chandler

______________________________________________

Dr. A. Gordon Emslie

______________________________________________
Dean of the Graduate College
Process Synchronization is the set of techniques that are used to coordinate execution amongst processes. A common resource such as shared memory or a device may require exclusive access. In a multitasked/multithreaded system, processes have to coordinate amongst themselves to ensure that access is exclusive and fair. The mutual exclusion problem is one of the problems in inter-process communication that has been studied extensively. A number of mutual exclusion algorithms have been proposed. They have had varying degrees of success in handling the problem. These algorithms can be classified into hardware solutions and software solutions. The most well-known software solutions are turn-taking algorithms.

This thesis gives an alternative mutual exclusion algorithm that is based on request and fair access instead of turn taking. The algorithmic details and the implementation of the new mutual exclusion algorithm are given in this thesis report. The new algorithm’s implementation was tested on different multi-tasking multi-processor systems. The test environments consisted of a Dual CPU Dell PowerEdge 1800 running Windows 2003 Advanced Server and a four CPU HP ProLiant DL580 running Windows 2000 Server. The new algorithm was implemented in C#. The new algorithm was tested using classical dining philosophers and race condition problems. The test results showed that the new algorithm successfully solved these traditional mutual exclusion problems.
ACKNOWLEDGEMENTS

I would like to express my appreciation to my graduate advisor Dr. M. H. Samadzadeh for his advisement, guidance, instruction, and encouragement throughout my thesis research work. His inspiring insight and patience guided me through my research and provided me with a valuable experience.

My sincere appreciation also extends to Drs. G. E. Hedrick and J. P. Chandler for their advice and serving on my graduate committee.

Finally, I also wish to thank my wife and parents for their encouragement and support throughout my graduate studies.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>I. INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>II. LITERATURE REVIEW</td>
<td>4</td>
</tr>
<tr>
<td>2.1 Control Problems of Concurrent Processes</td>
<td>4</td>
</tr>
<tr>
<td>2.1.1 Classification</td>
<td>4</td>
</tr>
<tr>
<td>2.1.2 Mutual Exclusion Algorithm Correctness</td>
<td>5</td>
</tr>
<tr>
<td>2.1.3 Classic Synchronization Problems</td>
<td>6</td>
</tr>
<tr>
<td>2.2 Well-Known Synchronization Solutions</td>
<td>7</td>
</tr>
<tr>
<td>2.2.1 Hardware Solutions</td>
<td>8</td>
</tr>
<tr>
<td>2.2.2 Software Mutual Exclusion Algorithms</td>
<td>8</td>
</tr>
<tr>
<td>2.3 Related work</td>
<td>14</td>
</tr>
<tr>
<td>III. THE NEW MUTUAL EXCLUSION ALGORITHM</td>
<td>16</td>
</tr>
<tr>
<td>3.1 Execution Order Theory</td>
<td>16</td>
</tr>
<tr>
<td>3.2 Assumptions</td>
<td>17</td>
</tr>
<tr>
<td>3.3 The New Mutual Exclusion Algorithm</td>
<td>18</td>
</tr>
<tr>
<td>3.3.1 Algorithm Design Considerations</td>
<td>18</td>
</tr>
<tr>
<td>3.3.2 Algorithm Specification</td>
<td>18</td>
</tr>
<tr>
<td>3.3.3 Informal Correctness Proof</td>
<td>22</td>
</tr>
<tr>
<td>3.4 The Advantage and Drawback of New Mutual Exclusion Algorithm</td>
<td>23</td>
</tr>
<tr>
<td>VI. IMPLEMENTATION AND TESTING</td>
<td>25</td>
</tr>
<tr>
<td>4.1 Implementation and Testing Environment</td>
<td>25</td>
</tr>
<tr>
<td>4.2 Basic Test Case for Race Condition</td>
<td>25</td>
</tr>
<tr>
<td>4.3 Using New Algorithm to Solve Traditional Mutual Exclusion Problem</td>
<td>31</td>
</tr>
<tr>
<td>V. SUMMARY, CONCLUSION, AND FUTURE WORK</td>
<td>34</td>
</tr>
<tr>
<td>5.1 Summary and Conclusion</td>
<td>34</td>
</tr>
<tr>
<td>5.2 Future Work</td>
<td>35</td>
</tr>
<tr>
<td>REFERENCES</td>
<td>36</td>
</tr>
<tr>
<td>APPENDICES</td>
<td>39</td>
</tr>
<tr>
<td>APPENDIX A: GLOSSARY</td>
<td>40</td>
</tr>
<tr>
<td>APPENDIX B: CODE LISTINGS</td>
<td>41</td>
</tr>
<tr>
<td>APPENDIX C: RESULTS OF TESTING</td>
<td>47</td>
</tr>
</tbody>
</table>
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Switch Write Pointer</td>
<td>20</td>
</tr>
</tbody>
</table>
## LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Ten Threads Update Shared Variable 5 times</td>
<td>30</td>
</tr>
</tbody>
</table>
CHAPTER I

INTRODUCTION

The mutual exclusion problem is a classic problem in operating systems. It is especially important in multi-tasked shared resource systems and distributed systems.

In a multi-process/multi-threading system, the set of mechanisms used to control the concurrent and parallel execution of the applications is generally called synchronization [Samadzadeh and Avutu 98]. Based on the degree of awareness of processes of one another, the relationship among concurrent processes can be classified as competition, cooperation by sharing, or cooperation by communication [Samadzadeh and Avutu 98].

There are hardware and software solutions to the mutual exclusion problem. Hardware solutions are typically based on having an atomic instruction such as the test-and-set instruction or the lock instruction. Hardware solutions have obvious limitation due to their dependence on the host architecture’s instruction set. Software solutions enforce mutual exclusion by using algorithms. They could generally be ported to different platforms.

Historically, the first correct mutual exclusion software solution was Dekker’s algorithm [Dijkstra 65a]. Other well-known algorithms are Peterson’s algorithm, Eisenberg and McGuire’s algorithm, and Lamport’s Algorithm [Lamport 87]. These algorithms have varying degrees of success in handling the problem with fairness. Most of the software solutions to the mutual exclusion problem are turn taking algorithms, where every process must follow the turn taking pattern and be aware of the current
status of other processes. The algorithms generally need to keep track of other process’ status and turn. Classified based on degree of awareness of other processes, these algorithms fall into the “cooperation by sharing” category.

The new mutual exclusion algorithm, which was designed, implemented, and tested as part of this thesis work, is also a software solution. It can be used to solve the n-processes mutual exclusion problem. The new algorithm is based on “request” rather than “request and turn”. Processes have an equal chance to gain access to the shared variable(s). The new algorithm solves the mutual exclusion problem by leveraging variable reference or pointer, which is available in most of modern object-oriented languages such as C++, JAVA, and C#. The new algorithm randomly chooses a “winner” process from among concurrent processes. Therefore, the selection is considered to be based on fair access. From the point of view of classification of process relationships, the new mutual exclusion algorithm belongs to the “competition” category. Processes are not aware of the existence of other processes, and there is no explicit inter-process communication amongst the participants in the competition.

Chapter II of this thesis gives a literature review. It discusses the concurrent processes’ control problems and relationships, and a number of solutions to the mutual exclusion problem. It reviews well-known software mutual exclusion algorithms along with the related work on Samadzadeh and Avutu’s mutual exclusion primitive [Samadzadeh and Avutu 98].

Chapter III provides a detailed discussion of the new mutual exclusion algorithm. It also addresses Lamport’s process execution order theory [Lamport 86a], design consideration, and specification.

Chapter IV describes the implementation of the new algorithm to solve classic mutual exclusion problems. It presents several test cases run in a multi-processor, multi-tasked environment and provides test results and analysis.
Chapter V presents the study summary and the future work.
CHAPTER II

LITERATURE REVIEW

2.1 Control Problems of Concurrent Processes

In modern multi-tasking and multi-processing systems, resources are shared among multiple processes or threads, and jobs are frequently switched to provide quick response times to interactive users. The OS must manage contention, synchronization, and communication among tasks as well as assign tasks to the available processor(s).

Synchronization basically means constraining the order in which things can happen in different processes. A major purpose of synchronization is to eliminate the possibility of a race condition. A race condition is a situation when the order of execution of two or more events can impact the outcome. Two or more processes are “racing” to get to the shared variable, only one of them wins but we cannot predict the winner in advance. Synchronization mechanisms can be classified into busy-wait and scheduler-based implementations. A busy-wait implementation is generally the better choice when the expected time a process has to wait is less than the time required to switch to a different process and back.

2.1.1 Classification

Based on the degree of awareness of one another, the relationship among concurrent processes can be classified as competition, cooperating by sharing, or cooperation by communication [Samadzadeh and Avutu 98]. Hence we can have competing as well as cooperating relationships among processes. Processes compete
with one another by attempting to access a system resource exclusively. In this relationship, processes are unaware of the existence of other processes. If two or more processes concurrently gain access to the same resource, conflict may result and the integrity of the data may be compromised. Processes can cooperate with one another by sharing information or message passing. In this case, processes are aware of the existence of other processes. In either relationship, i.e., competitive or cooperative, mutual exclusion needs to be enforced.

2.1.2 Mutual Exclusion Requirements

Any correct mutual exclusion solution must satisfy the following three requirements [Silberschatz et al. 05]:

1. Mutual Exclusion: If process one is executing in its critical section, no other processes can be executing in their critical sections.

2. Progress: If there is no process is in the critical section and some processes want to enter the critical section, only those processes that want to enter and are not in their remainder section can participate in the decision of which will enter their critical section next. And the decision cannot be postponed indefinitely.

3. Bounded Waiting: There must be a bound on the number of times that other processes are allowed to enter the critical section while a particular process is waiting to enter.

Early research in mutual exclusion was generally devoted to finding how to implement mutual exclusion algorithmically with no specific primitives. Dekker is generally credited with finding the first correct solution for two processes in the early 1960s [Dijkstra 65a]. Dijkstra published a version that works for N processes in 1965 [Dijkstra 65b]. Peterson published a simpler version of the two-process algorithm in 1981 [Peterson 81].
When trying to achieve mutual exclusion, one needs to prevent introducing deadlock and starvation. This is related to the second and third requirements. A process is deadlocked when it is waiting for some event that will never occur.

There are four necessary conditions for deadlock [Silberschatz et al. 05]:

1. Mutually Exclusive Access: At least one resource is non-sharable.
2. Hold and Wait: At least one process is holding some resources and waiting for other resources.
3. No Preemption: Resources cannot be taken away from a process that holds them.

All of the above four conditions are necessary for deadlock to occur. Thus, we can prevent deadlock by removing any one of them.

During the enforcement of mutual exclusion and elimination of deadlock, starvation can sometimes be introduced as a side effect. If starvation, which is unbounded waiting, is limited to a reasonable level and fairly so (i.e., for all processes), the solution is usually acceptable.

2.1.3 Classic Synchronization Problems

A number of synchronization problems occur frequently among cooperating processes. Those traditional problems have been researched extensively. Moreover, they have become a test bed for new mutual exclusion algorithms. Three of these common and classic problems are briefly described below.

1. Critical Sections Problem: A critical section is a code segment in a program in which some shared resource is referenced. During the execution of a critical section, mutual exclusion with respect to the respective resource must be ensured.

2. Producer/Consumer Problem: In its general form, a set of producer processes supplies messages to a set of consumer processes. They all share a common pool of buffers into which messages may be placed by producers and out of which they may be
removed by consumers. Producers must be prevented from overwriting a message and consumers must be prevented from attempting to remove non-existent messages.

3. Dining Philosophers Problem: Five philosophers sit around a table [Silberschatz et al. 05]. They alternate between thinking and eating. There is a rice plate for each philosopher on the table. When a philosopher wishes to eat, he/she picks up two chopsticks next to his/her plate. However, there are only five chopsticks on the table. Therefore, a philosopher can eat only when neither of his/her neighbors is eating. It is difficult to write a deadlock-free solution by using simple semaphores.

There are also other common synchronization problems such as the reader/writer problem and the cigarette smoker’s problem. These problems have been broadly used as models to test proposed mutual exclusion algorithms for correctness and effectiveness.

After introducing the new mutual exclusion algorithm in Chapter III, the algorithm is used to solve some of these typical synchronization problems in Chapter IV.

2.2 Well-Known Synchronization Solutions

2.2.1 Hardware Solutions

In the earlier years, hardware designers began to build read-modify-write instructions into their machines to achieve atomic operations. Common hardware read-modify-write instructions are Test-And-Set, Fetch-And-Add, Load-Linked/Store-Conditional, and Memory-Register-Exchange [Bershad et al. 92]. A simple implementation of mutual exclusion with Test-And-Set is given below.

\[
\begin{align*}
\text{Initial:} \\
\text{Boolean lock = false;} \\
\text{For Process i:} \\
\text{do} \{ \\
\quad \text{While(TestAndSet(lock));} \\
\quad <\text{critical section}> \\
\quad \text{Lock=false;} \\
\quad <\text{remainder section}> \\
\}
\end{align*}
\]
In general, a hardware-based algorithm performs more efficiently than a software-based algorithm due to support from atomic instructions. However, software-based algorithms have the advantage of flexibility, which can be used to take advantage of the structure of different architectures and systems [Zhang et al. 96]. Hardware mutual exclusion solutions are too low level to be used by an application. Another drawback is that the hardware instructions are platform specific. Usually they are used to implement OS-level primitives. However at the lowest level, typically higher level synchronization primitives are implemented using the hardware solutions.

2.2.2 Software Mutual Exclusion Algorithms

Dekker is generally credited with finding the first correct software solution to the two-process mutual exclusion problem [Silberschatz et al. 05]. This algorithm was discussed by Dijkstra in his classical paper published in 1965 [Dijkstra 65a]. Dijkstra also published a version that works for N processes [Dijkstra 65b]. Dekker’s algorithm for N processes is given below.

INITIALIZATION:

typedef char boolean;
...
shared boolean flags[n -1];
shared int turn;
...
turn = i ;
...
flags[i] = FREE;
...
flags[j] = FREE;
...
ENTRY PROTOCOL (for Process i):
/* claim the resource */
flags[i] = BUSY;

/* wait if another process is using the resource */
while (flags[j] == BUSY) {
/* if waiting for the resource, also wait for turn */
if (turn != i) {
/* but release the resource while waiting */
    flags[i] = FREE;
    while (turn != i) {

}
flags[i] = BUSY;
}

We can see from this algorithm that no process will enter its critical section without setting its flag. Every process checks the flags of others after setting its own. If all flags are set, the turn variable is used to allow only one process to proceed so that the mutual exclusion requirement is assured.

Peterson published a simpler version for two processes in 1981 [Peterson 81], as given below.

type pid = 1..2
turn : pid          // initial value doesn't matter
c : array [pid] of Boolean      // initialized to false
processor local i, otherguy : pid

procedure acquire
    c[i] := true
    turn := otherguy
    repeat until (not c[otherguy]) or (turn = i)

procedure release
    c[i] := false

Peterson also showed how to generalize his solution to N processes as given below [Peterson 81]. The algorithm takes O (log N) time to get somebody into the critical section.

INITIALIZATION:
    typedef char boolean;
    ...
    shared boolean flags[n -1];
    shared int turn;
    ...
    turn = i;
    ...
    flags[i] = FREE;
    ...
    flags[j] = FREE;
    ...
ENTRY PROTOCOL (for Process i):
/* claim the resource */
flags[i] = BUSY;

/* give away the turn */
turn = j;
/* wait while another process is using the resource *and*
has the turn */
while ((flags[j] == BUSY) && (turn != i)) {
  <critical section>
}

EXIT PROTOCOL (for Process i):
/* release the resource */
flags[i] = FREE;

Based on correct mutual exclusion solution requirements mentioned in Section 2.1.2., we can exam how Peterson’s algorithm meets those mutual exclusion solution requirements:

1) Mutual exclusion is assured: Assume there are two or more processes in their critical section. Since only one can have the turn, others must have reached the while test before the process with the turn set its flag. However, after setting its flag, other processes have to give away the turn. Other processes at the while test have already changed the turn and will not change it again, contradicting the assumption.

2) Progress requirement is assured: The turn variable is only considered when there are two or more processes trying to access the shared resource.

3) Bounded waiting is assured: When a process, which has exited the critical section reenters, it will give away the turn. If there are other processes waiting, one of the waiting processes will be the next to proceed.

Eisenberg and McGuire’s algorithm [Eisenberg and McGuire 72] (given below) uses a state flag to indicate each process state as being idle, waiting, or active. Each process scans all other processes only when all other processes are idle and the current process has the turn, then it can claim the shared resource.

INITIALIZATION:

shared enum states {IDLE, WAITING, ACTIVE} flags[n -1];
shared int turn;
int index; /* not shared! */
...
turn = 0;
...
for (index=0; index<n; index++) {
    flags[index] = IDLE;
}

ENTRY PROTOCOL (for Process i):
repeat {
    /* announce the need for the resource */
    flags[i] = WAITING;
    /* scan processes from the one with the turn up to this one */
    /* repeat if needed until the scan finds all processes idle */
    index = turn;
    while (index != i) {
        if (flags[index] != IDLE) index = turn;
        else index = index+1 mod n;
    }
    /* now tentatively claim the resource */
    flags[i] = ACTIVE;
    /* find the first active process besides this one, if any */
    index = 0;
    while ((index < n) && ((index == i) || (flags[index] != ACTIVE))) {
        index = index+1;
    }
    /* if there were no other active processes, AND if this one
    has the turn or else whoever has it is idle, then proceed;
    otherwise repeat the whole sequence */
    } until ((index >= n) && ((turn == i) || (flags[turn] == IDLE)));
    /* claim the turn and proceed */
    turn = i;

EXIT PROTOCOL (for Process i):
/* find a process that is not IDLE */
/* (if there are no others, this one is a candidate) */
index = turn+1 mod n;
while (flags[index] = IDLE) {
    index = index+1 mod n;
}
/* give the turn to a process that needs it, or keep it */
turn = index;
/* finished */
flag[i] = IDLE;
In 1987, Lamport published the first N-process solution [Lamport 87] (outlined below) that requires constant time per acquisition in the absence of contention.

```plaintext
type pid = 1..N
x, y : pid                      // initialized to 0
b : array [pid] of Boolean      // initialized to false

procedure acquire

loop
  b[i] := true
  x := i               // most recent process to begin protocol
  if y <> 0
    b[i] := false
    repeat until y = 0
    continue
  y := i
  if x <> i               // competition: recover
    b[i] := false
    for j <> i in pid
      repeat until not b[j]
      if y <> i
        repeat until y = 0
        continue
  return

procedure release

y := 0
b[i] := false
```

Usually these well-known algorithms require a process turn number and a status flag. So each process needs to know the status of the other processes.

Semaphores and Monitors as synchronization tools have been broadly used in modern operating systems and multi-process/multi-threaded applications. They are operating system level library primitives. To use at the application level, they both require language and compiler support. The semaphore was suggested by Dijkstra in 1965 [Dijkstra 65a]. Dijkstra originally proposed binary semaphores.

A binary semaphore is basically all that is needed for mutual exclusion. It is almost always initialized to 1 and its value should never exceed 1. Acquire_mutex is denoted by P and Release_mutex is denoted by V.
Here is the general semaphore semantics:

\[
P(S): \text{if } S \geq 1 \text{ then} \\
S:S-1 \\
\text{else} \\
\text{the executing process places itself in } S\text{’s waiting} \\
\text{queue and release the CPU by invoking the CPU scheduler} \\
\text{endif} \\
\]

\[
V(S): \text{if } S\text{’s waiting queue is not empty then} \\
\text{move one process from the waiting queue to the ready} \\
\text{queue and invoke the CPU scheduler} \\
\text{else} \\
S:=S+1 \\
\text{endif} \\
\]

A general semaphore can be used to solve typical synchronization problems such as the bounded buffer problem, as given below.

\[
\text{shared buf : array } [1..\text{SIZE}] \text{ of data} \\
\text{shared next_full, next_empty : integer } := \text{1} \\
\text{shared mutex : semaphore } := \text{1} \\
\text{shared empty_slots, full_slots : semaphore } := \text{SIZE, 0} \\
\]

\[
\text{procedure insert (d : data) :} \\
\text{\hspace{1cm} P (empty_slots)} \\
\text{\hspace{1cm} P (mutex)} \\
\text{\hspace{1cm} buf[next_empty] := d} \\
\text{\hspace{1cm} next_empty := next_empty mod SIZE + 1} \\
\text{\hspace{1cm} V (mutex)} \\
\text{\hspace{1cm} V (full_slots)} \\
\]

\[
\text{function remove returns data :} \\
\text{\hspace{1cm} P (full_slots)} \\
\text{\hspace{1cm} P (mutex)} \\
\text{\hspace{1cm} d : data := buf[next_full]} \\
\text{\hspace{1cm} next_full := next_full mod SIZE + 1} \\
\text{\hspace{1cm} V (mutex)} \\
\text{\hspace{1cm} V (empty_slots)} \\
\text{\hspace{1cm} return d} \\
\]

The Monitors was developed by Brinch Hansen [Brinch Hansen 73] and the complete definition was given by Hoare [Hoare 74]. Monitors provide a structured concurrent programming primitive, that can be used by processes to ensure exclusive access to resources, as well as for synchronization and communication among processes. A monitor module typically encapsulates both the resource definition and the operations/procedures that exclusively manipulate it. Those procedures are the gateway to the shared resource and are called by other processes to access the resource. Only one
call to a monitor procedure can be active at a time. This restriction protects the data inside the monitor from simultaneous access by multiple users. Thus, mutual exclusion is easily enforced among tasks using a monitor. Processes that attempt monitor entry while the monitor is occupied by another process are blocked and join a monitor entry queue.

To synchronize tasks within the monitor, a condition variable is used to temporarily delay processes executing in a monitor. A condition variable may be declared only within a monitor and has no numeric value like semaphores do. Two operations, \textit{wait} and \textit{signal}, are defined on condition variables [Silberschatz et al. 05]. The \textit{wait} operation suspends/blocks the execution of a calling process until a certain condition becomes true. Then the monitor allows another task to enter. The execution of a signal operation causes the condition to become true. Then some process that had been suspended after a \textit{wait} on that condition becomes eligible for execution. If there are no waiting processes, the \textit{signal} operator is ignored.

A condition variable is associated with a queue of the processes that are currently waiting on that condition. The first-in-first-out (FIFO) queuing discipline is generally used with these queues, but priority queues can also be implemented by specifying the priority of the process to be delayed as a parameter in the \textit{wait} operation. Condition variables are assumed to be fair in the sense that a process will not remain suspended forever on a condition variable.

2.3 Related Work

In 1998, Samadzadeh and Avutu proposed a new mutual exclusion primitive [Samadzadeh and Avutu 98]. This primitive was designed to be easy to understand and a fair and an efficient solution for the synchronization problem at a higher level than semaphores. This primitive uses process priorities. In this synchronization primitive, the
arriving processes are classified based on their priority. The synchronization agents are also divided into high priority and low priority agents. A queue is associated with each agent. A subset of all of the available synchronization agents is designated as “active”. The remaining agents are inactive. Some of the active agents are high priority agents, and the remaining ones are low priority agents. When an arriving process enters the system, its priority is checked and it is assigned an agent if one is free. Otherwise, the process is put into the shortest queue according to its priority. If higher priority agents are free, they could serve the lower priority queues. In case all agents are busy and all queues are full, the incoming process will wait in an overflow queue. When the overflow queue is full, the process will not be allowed to enter the synchronization system. The synchronization system is smart enough to allocate agents, to put processes in the appropriate queues, to activate or deactivate agents, and to control overflows.

Samadzadeh and Avutu provided two implementations for their mutual exclusion primitive: an atomic-signal/atomic-wait implementation and a semaphore implementation.
CHAPTER III

THE NEW MUTUAL EXCLUSION ALGORITHM

3.1 Execution Order Theory

In 1986 Lamport published a scheme about execution order [Lamport 86a].

\[
A \rightarrow B = \forall a \in A : \forall b \in B : a \to b
\]

\[
A \cdots \rightarrow B = \exists a \in A : \exists b \in B : a \to b \text{ or } a = b
\]

An execution consists of a nonempty set of space-time events. \( A \rightarrow B \) means that every event of A happened earlier than every event of B. So, \( A \rightarrow B \) stands for “A precedes B”. \( A \cdots \rightarrow B \) means that some event of A either precedes or is the same as some event of B. So, \( A \cdots \rightarrow B \) stands for “A can causally affect B”. In higher-level operation, where sets of lower-level events are involved, this relation is maintained.

\[
R \rightarrow S = \forall A \in R : \forall B \in S : A \rightarrow B
\]

\[
R \cdots \rightarrow S = \exists A \in R : \exists B \in S : A \cdots \rightarrow B \text{ or } A = B
\]

In the proposed algorithm, each process performs three operations: 1) write process ID to a shared ID holder variable through a write pointer (also called process ID writer), 2) redirect the write printer (PID writer) so that no process can use it to write to the ID holder variable, and 3) check the ID holder variable to determine which process can enter critical section. Among processes, the order of execution follows the formula as shown above. It is assured that operation 2 precedes operation 3. When operation 3 execute, operation 2 must have been completed by one of the participating processes.
3.2 Assumptions

In a basic CPU instruction set, it is assumed that primitive instructions such as load, store, and test are atomic instructions [Silberschatz et al. 05]. Therefore, in the proposed algorithm, operations “read from” and “write to” a shared variable are considered to be executing atomically. However, “read-modify-write” is not considered an atomic operation. These are analogous to the assumptions made in Dijkstra’s algorithm [Dijkstra 65].

Also, as Lamport stated [Lamport 86b], the mutual exclusion problem has a number of different requirements. In the basic requirements, it is assumed that each process’ program contains a non-critical section and a critical section. These sections are executed alternately. So, process $i$ would generate following execution sequence.

\[
\text{NCS}[1]i \rightarrow \text{CS}[1]i \rightarrow \text{NCS}[2]i \rightarrow \text{CS}[2]i \rightarrow \ldots
\]

NCS[k]i denotes the kth execution of process i’s non-critical section. It is assumed that the CS[k] are terminating operations, which means that a process never “halts” in its critical section. However, NCS[k]i may be a non-terminating operation for some k.

To implement mutual exclusion, we must add some synchronization operations to process $i$. Therefore, process $i$ could generate the following sequence of operation executions [Lamport 86b].

\[
\text{NCS}[1]i \rightarrow \text{trying}[1]i \rightarrow \text{CS}[1]i \rightarrow \text{exit}[1]i \rightarrow \text{NCS}[2]i
\]

In this partial execution segment, trying[1]i and exit[1]i denote the operation executions generated by the first time executions of “trying” and “exiting” which indicate gaining access to the critical section and relinquishing the critical section, respectively.
3.3 The New Mutual Exclusion Algorithm

3.3.1 Algorithm Design Considerations

The new algorithm is designed to solve the classical mutual exclusion problem. To access a shared variable exclusively, each competing process will have three sections: trying section, critical section, and exit section. During the trying section, which is the competition phase, a process tries to gain access. During the critical section, the process manipulates the shared variable. During the exit section, the process returns the shared variable to the available state and another competition begins.

To achieve the mutual exclusion objective, we need a competition mechanism to ensure that no matter how many processes participate in the competition, one and only one of them would be the winner. The key is to control when the competition finishes and when it starts again.

3.3.2 Algorithm Specification

The purpose is to select a winner to gain access to the shared resources. This algorithm is analogous to prize drawing games. Suppose we have a crystal box and there is a slot on top of box. Everyone participating in the competition has a unique ID. They throw their ID card into the crystal box through the slot. After a random period, the slot will be closed. Then this round of competition finishes. The winner’s ID card stays on top and is visible in the crystal box, then the winner can access the shared resources. There are three distinct situations for all participants: 1) the one whose ID card is on top, 2) those participants whose cards are in the box but are covered by others’ ID cards, and 3) those who tried to threw in their cards after the slot was closed (their cards are outside the box). Group 2 and group 3 participants will wait for the next round when a new box will be set up by the previous winner.

The CrystalBox data structure is defined below.
The CrystalBox data structure is composed of a winnerPID variable that stores the winner’s process ID, a loserPID variable that stores the loser’s process ID, and a reference/pointer type variable pIDWriter. The pIDWriter is a process ID writer/write pointer, that can reference/point to winnerPID or loserPID. By default, pIDWriter reference winnerPID. All processes that want to write their IDs to the CrystalBox data structure must go through pIDWriter.

The new algorithm is specified below.

Initialize:
1. var crystal_box := new CrystalBox()

For Process i:

trying Section:
While(true)
{
2. var myBoxRef = crystal_box
3. myBoxRef.pIDWriter.pID = my Process ID
4. myBoxRef.pIDWriter = myBoxRef.loserPID
5. If( myBoxRef.winnerPID.pID == my process ID )
6.    go to Critical Section
7. Else
8.    Sleep(Random)  //release CPU, and try it later
}

Critical Section:
9. Use the Shared Resources

Exit Section:
10. crystal_box:=new CrystalBox()
In the line 1 of program initialization section, a CrystalBox instance called crystal_box is created. It is a global shared variable, which can be accessed by all processes. By default crystal_box.pIDWriter references crystal_box.winnerPID.

For each process $i$, in its trying block, there is a try loop. The process keeps trying until it gains access. On line 2, process $i$ defines crystal_box’s local reference myBoxRef. This reference is used as a local alias to access the shared data structure crystal_box’s members.

In line 3, process $i$ writes its process ID to the crystal_box data structure by using the process ID writer in crystal_box.

![Switch Write Pointer](image)

**Figure 1. Switch Write Pointer**

In line 4, process $i$ disassociates the crystal_box reference variable pIDWriter from the winnerPID (see Figure 1). Instead of having pIDWriter reference winnerPID, process $i$ sets pIDWriter to point to loserPID now. So after this instruction is performed, no other processes can write their process ID to winnerPID in the crystal_box data structure. In other words, competition is over and a winner has been selected. This line is the most important part in the algorithm.
In line 5, process $i$ checks the winning process ID in crystal_box to determine if it is the winner.

If process $i$ is the winner, it will break out from the loop at line 6 and enter the critical section. Otherwise, in line 8, it will sleep for a random period of time to allow the winning process to finish its job in the critical section. Afterwards, the process will try again in the loop to enter the critical section.

When the winner completes its critical section, it will enter the exit section. The winning process will recreate the crystal_box data structure in line 10. Since other processes may still hold reference to the old crystal_box data structure, reusing the crystal_box data structure and resetting pIDWriter may cause a race condition in the trying block. By default, the newly created crystal_box’s pIDWriter is associated with winnerPID.

Three scenarios may occur among processes.

1) When process $i$ executes line 3, another process has completed line 4. The PIDwriter variable has been diverted to the loserPID variable. Therefore, process $i$ could not write its ID into the winnerPID variable.

2) When process $i$ write to the data structure, the PIDwriter is still referencing winnerPID. However, process $j$ arrives later and writes to the data structure just before the PIDwriter is disassociated from winnerPID by another process. Then, process $i$’s process ID is overwritten by process $j$, and process $j$ wins the access.

3) Process $i$ is the last one writing to the data structure just before another process disassociates the PIDwriter from the winnerPID variable. Process $i$ is the winner in this case.

Note that only the first disassociation operation takes effect. Other disassociation operations are redundant. Nevertheless, it is a necessary step to guarantee that a winner in the crystal_box data structure has been finalized.
After a round of selection, the crystal_box data structure and the PIDwriter variable cannot be reused for a while. To reuse the crystal_box, we need to reset PIDwriter to reference winnerPID. If crystal_box is still being held by some process, and the reset operation occurs between lines 4 and 5, then the algorithm will fail. Therefore, at the end of the exit section a new instance of the crystal_box data structure is created for the next round competition.

Most current high-level languages such as Java and C# have garbage collection mechanisms. The garbage collection process can recycle these unused crystal_box data structures only when no one has a reference to them.

Any language that supports the pointer type or reference type can be used to implement this algorithm.

3.3.3 Informal Correctness Proof

As mentioned in Section 2.1.2, a correct mutual exclusion solution needs to satisfy three requirements: 1. mutual exclusion, 2. progress, and 3. bounded waiting. Based on these requirements, we can informally prove that the new mutual exclusion algorithm is correct, as outlined below.

1) The mutual exclusion requirement is assured. Each winner process selection starts with a new crystal_box data structure. Each process gets a local reference to crystal_box. In a process, the three steps of using PIDwriter to write its PID, disassociating PIDwriter from the winnerPID variable, and checking the winnerPID value, are executed in order. This guarantees that PIDwriter has been disassociated from the winnerPID variable when a process checks the winnerPID. Therefore, there is one and only one winner process ID in the winnerPID variable. The winner process now gains access to the shared resource exclusively.

2) The progress requirement is assured. If there is no other process requesting the shared resource, the current process is guaranteed to be the winner and to gain access.
3) Bounded waiting is assured. In the new algorithm every process has an equal chance to enter the critical section. For an N-process system, the worst case is when a process wait N turns to enter its critical section.

3.4 Advantages/Drawbacks of the New Mutual Exclusion Algorithm

This new algorithm has the following advantageous characteristics.

1) This algorithm can be implemented in any language that supports a pointer or reference data type. For this thesis, all test programs were written in C#.

2) The new mutual exclusion algorithm has cross platform portability. It is an application level algorithm. It does not depend on a particular platform, operating system, or runtime support library.

3) This algorithm is autonomous. Each process does not need to know how many processes are in the system, and does not need to trace the other processes’ states. There is no direct inter-process communication.

4) With garbage collection, the new algorithm only needs a constant number of variables (crystal_box data structure). Most modern language have a garbage collection mechanism. In theory, this algorithm only needs one data structure (called crystal_box) for each round of selections. However, that data structure cannot be reused in next selection, because some process may still be holding the old crystal_box. Garbage collection is a separate process typically supported by the implementation language. It can recycle the allocated memory when there is no reference to it. So in this scenario, only a constant number of data structure should be in use at a time. In the worst case without garbage collection, this algorithm will need N data structures for an N process system.
5) Every process gets an equal chance to access the shared resource. This algorithm is not a turn taking or a flag-controlled algorithm. It does not need a turn and a status tracing mechanism. The winner is randomly generated. It appears to be similar to real world situations.

   This new algorithm has the following drawbacks.

1) Since the winner is “randomly” generated, in some cases it may generate starvation.

2) Since all waiting processes have an equal chance to win the election, if FCFS needs to be enforced at this level, this algorithm may not be the best choice. We can use a higher-level primitive or a queuing mechanism to overcome this disadvantage.
CHAPTER IV
IMPLEMENTATION AND TESTING

4.1 Implementation and Testing Environment

To test the new algorithm, a series of test programs were written in a multi-tasking environment. A Dell PowerEdge 1800 server and HP ProLiant DL580 are used to run these test programs.

Dell PowerEdge 1800 has Dual Intel Xeon 3GHz processors and runs Windows 2003 advanced server operating system that supports multiple processors. The HP server has 4 Intel Xeon 2.8G CPUs running Windows 2000.

The new algorithm can be implemented in any language that supports a pointer or reference data type. The test program was implemented in C#.net. The C# language and the .NET framework provide a library and the necessary functionality to develop multi-threaded programs. C#.net also provides traditional synchronization methods such as monitors and locks. The test programs do not use any operating system level or language level synchronization function calls.

4.2 Basic Test Case for Race Condition

To test the new mutual exclusion algorithm, first it is used to examine the race condition problem.

The test case has 500 threads access a shared data. Each thread increases the shared data by 1. The shared data is initialized to 0.
To do the comparison, the first program shows the result of the potential race condition. In this program, the threads access the shared variable without any synchronization control.

```java
static int sharedData=0;
static public void AccessSD()
{
  int x;
  Thread.Sleep(0);
  x=sharedData;
  Thread.Sleep(1);
  sharedData=x+1;
}
```

A total of 500 threads were created to run the above function. Each thread ran the AccessSD function to increment sharedData by 1. The variable sharedData is a static class variable. Please note that in the AccessSD function, there are two sleep functions between reading of the shared variable and writing to it.

As expected, the result of shared data would be between 2 and 500. The test program was executed 400 times on the multiprocessor system PE 1800. The result was from 18 to 88. The result depends on the CPU speed and the actual sleep time between read and write.

The second program was implemented using the new mutual exclusion algorithm proposed in this thesis.

The following code segment sets up the basic data structure.

```java
class TID
{
  public int thID=-1;
}

class CrystalBox
{
  public TID winnerThread=null;
  public TID loserThread=null;
  public TID tIDWriter=null;

  public CrystalBox()
  {
    winnerThread=new TID();
    loserThread=new TID();
    tIDWriter=winnerThread;
  }
}
```
In the above code, the TID class is defined to hold the thread IDs. The CrystalBox class creates a crystal box which is the winnerThread data member and the write pointer called tIDWriter. The loserThread data member is just a container. When the write pointer switches from winnerThread to loserThread, the late coming thread IDs are redirected to the loserThread variable.

The following code segment creates multiple threads.

```csharp
for (int i = 0; i < numThreads; i++)
{
    Thread myThread = new Thread(new ThreadStart(UseResource));
    myThread.Name = String.Format("Thread{0}", i + 1);
    myThread.Start();
}
```

The code segment below implements the new algorithm.

```csharp
private static void UseResource()
{
    int myID = AppDomain.GetCurrentThreadId();
    // Wait until it is safe to enter.
    while (true)
    {
        CrystalBox myBox = cBox;
        Thread.VolatileWrite(ref myBox.tIDWriter.thID, myID);
        myBox.tIDWriter = myBox.loserThread;
        //Thread.Sleep(0);
        if (myID == Thread.VolatileRead(ref myBox.winnerThread.thID))
        {
            break;
        }
        Thread.Sleep(1);
    }
    Thread.Sleep(0);
    int local = sharedResource;
    Thread.Sleep(1);
    sharedResource = local + 1;
    cBox = new CrystalBox();
}
```

Each thread uses the above function to access the shared variable “sharedResource”. In line 1, the thread gets its thread ID. Then, it enters the trying block in line 2. In the while loop, a thread will have its local reference to the global
variable cBox which is an instance of the CrystalBox class. In line 4, the thread tries to write its thread ID into the cBox’s winnerThread variable. Modern CPUs are usually build with multi-level caches to overcome the CPU and RAM speed difference. Data are pre-fetched from main memory into cache to satisfy the needs of the higher speed CPU. To avoid using a CPU register instruction cache, system provides Volatile Write and Volatile Read instructions. When the CPU runs these instructions, it will get the value from the memory address instead of using the pre-fetched data in cache. In line 4, Volatile Write is used. So that if the write pointer has just changed, the CPU will use the latest reference to the write pointer.

The next step is to switch the write pointer to the loserThread ID holder. This action prevents others from changing the winner value in the crystal box.

In line 6, the thread reads the winner ID from cBox and compares it with its own ID. If winner ID is not the current thread’s ID, the thread will sleep for a random time and will go back to line 3 and try again.

If this thread is a winner thread, then it will jump to line 9 to execute the critical section. In lines 10-12, the shared variable’s value is incremented. The same amount of sleep time as the race condition case is introduced between the read and the update to demonstrate the mutual exclusion result.

After the critical section, the thread creates a new cBox structure in line 13. In this new cBox structure, the write pointer points to the winnerThread variable. Therefore, in next round, the requesting threads can write their ID to the winnerThread to compete again.

This implementation was tested on Dell 1800 which has two Intel CPUs. About 10% of a typical test result is listed in Appendix C, List 1. In the test program, there are 500 threads. The test program was executed 400 times. As expected, the integrity of the shared variable was preserved and the value reported was consistently 500.
In a more complicated test case, the threads updated the shared variable by 1 five times in a loop. This implementation is similar to the above test case except that each process accesses the shared variable five times in a loop instead of just once. The detailed code listing is given in Appendix B. About 10% of the test result is listed in Appendix C, List 2.

```csharp
private static void MyThreadProc()
{
    int myID=Thread.CurrentThread.ManagedThreadId;
    for(int i = 0; i < 5; i++)
    {
        Console.WriteLine("Thread#{0} #{1} Time requests Resource.", myID, i);
        UseResource();
    }
}

// This method represents a resource that must be synchronized
// so that only one thread at a time can enter.
private static void UseResource()
{
    int myID = Thread.CurrentThread.ManagedThreadId;
    // Wait until it is safe to enter.
    while(true)
    {
        CrystalBox myBox=cBox;
        Console.WriteLine("\tThread#{0} is requesting Resource.", myID);
        Thread.VolatileWrite(ref myBox.tIDWriter.thID, myID);
        myBox.tIDWriter=myBox.loserThread;
        //Thread.Sleep(0);
        if(myID==Thread.VolatileRead(ref myBox.winnerThread.thID))
            break;
        Thread.Sleep(1);
    }
    Console.WriteLine("\tThread#{0} has entered the protected area", myID);
    int local=sharedResource;
    // Simulate some work.
    Thread.Sleep(20);
    sharedResource=local+1;
    Console.WriteLine("\tThread#{0} is leaving the protected area\r\n\tSharedResource={1}\r\n", myID, sharedResource);
    cBox=new CrystalBox();
}
```
Table 1 Ten Threads Update Shared Variable 5 Times

<table>
<thead>
<tr>
<th>Round No.</th>
<th>Winner Process ID</th>
<th>Shared Variable Value</th>
<th>Round No.</th>
<th>Winner Process ID</th>
<th>Shared Variable Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>6844</td>
<td>1</td>
<td>26</td>
<td>6844</td>
<td>26</td>
</tr>
<tr>
<td>2</td>
<td>6844</td>
<td>2</td>
<td>27</td>
<td>6420</td>
<td>27</td>
</tr>
<tr>
<td>3</td>
<td>6844</td>
<td>3</td>
<td>28</td>
<td>6328</td>
<td>28</td>
</tr>
<tr>
<td>4</td>
<td>6780</td>
<td>4</td>
<td>29</td>
<td>6328</td>
<td>29</td>
</tr>
<tr>
<td>5</td>
<td>6844</td>
<td>5</td>
<td>30</td>
<td>6780</td>
<td>30</td>
</tr>
<tr>
<td>6</td>
<td>6696</td>
<td>6</td>
<td>31</td>
<td>6780</td>
<td>31</td>
</tr>
<tr>
<td>7</td>
<td>6696</td>
<td>7</td>
<td>32</td>
<td>6780</td>
<td>32</td>
</tr>
<tr>
<td>8</td>
<td>6696</td>
<td>8</td>
<td>33</td>
<td>6372</td>
<td>33</td>
</tr>
<tr>
<td>9</td>
<td>6372</td>
<td>9</td>
<td>34</td>
<td>6372</td>
<td>34</td>
</tr>
<tr>
<td>10</td>
<td>6372</td>
<td>10</td>
<td>35</td>
<td>6372</td>
<td>35</td>
</tr>
<tr>
<td>11</td>
<td>6296</td>
<td>11</td>
<td>36</td>
<td>6420</td>
<td>36</td>
</tr>
<tr>
<td>12</td>
<td>6452</td>
<td>12</td>
<td>37</td>
<td>6420</td>
<td>37</td>
</tr>
<tr>
<td>13</td>
<td>6452</td>
<td>13</td>
<td>38</td>
<td>6420</td>
<td>38</td>
</tr>
<tr>
<td>14</td>
<td>6328</td>
<td>14</td>
<td>39</td>
<td>6384</td>
<td>39</td>
</tr>
<tr>
<td>15</td>
<td>6328</td>
<td>15</td>
<td>40</td>
<td>6696</td>
<td>40</td>
</tr>
<tr>
<td>16</td>
<td>6328</td>
<td>16</td>
<td>41</td>
<td>6420</td>
<td>41</td>
</tr>
<tr>
<td>17</td>
<td>6452</td>
<td>17</td>
<td>42</td>
<td>6292</td>
<td>42</td>
</tr>
<tr>
<td>18</td>
<td>6452</td>
<td>18</td>
<td>43</td>
<td>6292</td>
<td>43</td>
</tr>
<tr>
<td>19</td>
<td>6452</td>
<td>19</td>
<td>44</td>
<td>6292</td>
<td>44</td>
</tr>
<tr>
<td>20</td>
<td>6384</td>
<td>20</td>
<td>45</td>
<td>6292</td>
<td>45</td>
</tr>
<tr>
<td>21</td>
<td>6384</td>
<td>21</td>
<td>46</td>
<td>6292</td>
<td>46</td>
</tr>
<tr>
<td>22</td>
<td>6384</td>
<td>22</td>
<td>47</td>
<td>6296</td>
<td>47</td>
</tr>
<tr>
<td>23</td>
<td>6384</td>
<td>23</td>
<td>48</td>
<td>6292</td>
<td>48</td>
</tr>
<tr>
<td>24</td>
<td>6296</td>
<td>24</td>
<td>49</td>
<td>6780</td>
<td>49</td>
</tr>
<tr>
<td>25</td>
<td>6296</td>
<td>25</td>
<td>50</td>
<td>6696</td>
<td>50</td>
</tr>
</tbody>
</table>

Table 1 shows the ten threads test case. Each thread accesses the shared variable 5 times in a loop to increase its value by 1. The shared variable is initially set to 0. The final value should be 50. As before, to simulate the computational work that could be done in the critical section, a sleep time of 20 ms is used. This case tested fair access and bounded waiting. As the table shows, when one thread tries to enter the critical section and update the shared variable multiple times, it is allowed to enter and there is no apparent lock-step pattern of turn taking involved in accessing the shared variable. The test case runs using 10 threads, 20 threads, 30 threads, 40 threads, and 50 threads. All results show the shared variable with correct value.
4.3 Using the New Algorithm to Solve a Traditional Mutual Exclusion Problem

The dining philosophers problem is one of the traditional mutual exclusion problems. In the following test program, a new algorithm is given to solve this problem. In this implementation, there are 5 philosophers #0 through #4 and 5 chopsticks #0 through #4. Each philosopher $i$ needs chopsticks #$i$ and #$i+1\%5$ to eat. Philosopher $i$ can eat only when she gains access to both chopsticks. Otherwise, she has to release the one she has. Chopsticks #0 through #4 are shared variables in this problem.

In the implementation, the crystalBox data structure is established as follows.

```csharp
class TID
{
    public int id;
}
class CrystalBox
{
    public TID csOwner = new TID();
    public TID IDWriter = null;
    public TID nulll = new TID();
    public CrystalBox()
    {
        IDWriter = csOwner;
    }
}
```

Then, 5 crystalBox data structures are defined to represent the 5 chopsticks.

```csharp
static CrystalBox[] cBox = new CrystalBox[5];
```

The following code initializes these data structures and creates the 5 philosophers.

```csharp
for (int i = 0; i < 5; i++)
{
    cBox[i] = new CrystalBox();
}
Thread[] threads = new Thread[5];
for (int j = 0; j < 5; j++)
{
    Dph d = new Dph();
    threads[j] = new Thread(new ThreadStart(d.GetChopSticks));
    threads[j].Name = j.ToString();
    threads[j].Start();
}
```

The core function to solve the dining philosophers problem is given below.
public void GetChopSticks()
{
    int i = int.Parse(Thread.CurrentThread.Name);
    int j = (i + 1) % 5;
    int myID = Thread.CurrentThread.ManagedThreadId;
    for(;;)
    {
        CrystalBox myboxi=cBox[i];
        Thread.VolatileWrite(ref myboxi.IDWriter.id, myID);
        myboxi.IDWriter = myboxi.nill;
        if (myID==Thread.VolatileRead(ref myboxi.csOwner.id))
        {
            CrystalBox myboxj = cBox[j];
            Thread.VolatileWrite(ref myboxj.IDWriter.id, myID);
            Myboxj.IDWriter = myboxj.nill;
            if (myID == Thread.VolatileRead(ref myboxj.csOwner.id))
            {
                break;
            }
        }
        else
        {
            cBox[i] = new CrystalBox();
        }
    }
    //Eating
    Console.WriteLine("\nPhilosopher: {0} is eating", i);
    Thread.Sleep(5);
    Console.WriteLine("Philosopher: {0} is done", i);
    cBox[i] = new CrystalBox();
    cBox[j] = new CrystalBox();
}

For a philosopher i, chopsticks i and (i+1)%5 are both needed for eating. So, in a loop, thread i first tries to store its thread ID to cBox[i] as shown in line 7. Then the thread moves the IDwriter to nill. If thread i has the chopstick i’s ownership, it will compete for chopstick j. In a similar competition process, if thread i does not gain chopstick j=(i+1)%5, thread i need to release chopstick i. That is shown on line 18. If it gains both chopsticks, it will go to line 23 to enjoy the food. At the end of lines 26 and 27, the thread i will reset the crystal boxes i and j which would in turn release both chopsticks for others to use.

This test was executed 100 times on a 2 CPU system. Here are some typical results.
The test result shows that the average degree of concurrency is 2. There is no deadlock.

The new mutual exclusion algorithm can solve the dining philosophers problem. The algorithm did not limit the degree of concurrency. In addition, new algorithm prevents deadlock from occurring.
CHAPTER V
SUMMARY, CONCLUSION, AND FUTURE WORK

5.1 Summary and Conclusion

The research work reported in this thesis proposes a new mutual exclusion algorithm to solve the mutual exclusion problem. The new algorithm is an application level software solution, which is based on request and fair access instead of turn-taking. Chapter I gives a brief introduction to the new algorithm. Chapter II provides literature review to concurrent process control problems. Some well-known synchronization solutions are also reviewed in Chapter II. Chapter III gives the new mutual exclusion algorithm’s specification, and discusses its advantages and drawbacks. Chapter IV contains the new algorithm’s implementation and test results. The new algorithm is generic enough to be implemented in any language that supports the pointer and reference data type. The implementation of the new algorithm can be easily ported to other platforms.

The new algorithm has the following advantages:

1) The algorithm is based upon request. Every process that wants to access the shared resource has an equal chance of gaining.

2) Each process controls its own behavior to avoid the race conditions. The algorithm does not have a centralized arbitrator to control the traffic.

3) The new algorithm is not a turn-taking algorithm. It is based on fair access.

4) It does not need an inter-process communication mechanism. Each process does not need to know the other processes’ existence or states.
5) It is easy to understand and can be implemented in any languages that supports pointer and reference data types.

6) With a garbage collection mechanism, the global variable usage can be constant, otherwise it could be O(n) for n processes.

Some disadvantages of the new algorithm are given below:

1) This algorithm may generate some degree of starvation. Since the winner is randomly selected, the possibility of having a process starving exists but it is low.

2) It is a busy wait algorithm.

5.2 Future Work

In future research, the performance of new algorithm needs to be studied. The data needed to be collected are: performance measurement based on different platforms and host instruction sets, performance measurement based on the length of critical section, and performance measurement based on the degree of concurrency. With the performance data, the new algorithm could probably be optimized.

The degree of starvation needs to be further studied. How to reduce or prevent starvation is another area to future research.

In future work, the algorithm can also be improved by introducing a wait mechanism. Other processes may be put to sleep when a winner process is in the critical section. After the winner process completes its task, it could send a signal to wake up the waiting processes. Such a mechanism would make the algorithm less of a busy wait algorithm.
REFERENCES


[Peterson 81] G. L. Peterson, “Myths about the Mutual Exclusion Problem,”  

Exclusion Primitive: Simulation and Validation”, *The International Journal of  

[Silberschatz et al. 05] A. Silberschatz, P. Galvin, and G. Gagne, *Operating System  


Annual ACM Symposium on Principles of Distributed Computing*, pp. 159-168,  

[Zhang et al. 96] X. Zhang, Y. Yan, and Robert Castaneda, “Evaluating and Designing  
Software Mutual Exclusion Algorithms on Shared-Memory Multiprocessors”,  
APPENDICES
APPENDIX A

GLOSSARY

Critical-Section: A segment of code in a process that modifies the shared variable(s) and hence must be executed atomically.

Deadlock: A situation in which a process or thread is waiting for an event that will never occur and therefore cannot continue execution.

Priority: A measure of a process’ or thread’s importance used to determine the order and duration of execution.

Race Condition: A situation that occurs when multiple threads simultaneously compete for the same serially reusable resource, and that resource is allocated to these threads in an indeterminate order. This can cause subtle program errors when the order in which threads access a resource is important.

Semaphore: A mutual exclusion abstraction that uses two atomic operations (P and V) to access a protected integer variable that determines if threads may enter their critical sections.

Starvation: A situation in which a thread waits for an event that might never occur, also called indefinite postponement.

Synchronization: Coordination between asynchronous concurrent threads to sequentialize their access to shard resources.

Task: A light-weight process that shares address spaces with another light-weight process.
APPENDIX B

CODE LISTINGS

This appendix contains three code lists. They are used in Chapter IV to implement and test the new algorithm. These programs are written in C# and are compiled in Microsoft Visual Studio 2005.

1. The new mutual exclusion implementation for the critical section problem.

The test case has 500 threads that access the shared variable. Each thread increases the value of the shared variable by 1. The shared variable is initialized to 0.

```csharp
using System;
using System.Threading;

namespace CrystalBox
{
    class TID
    {
       public int thID = -1;
    }

    class CrystalBox
    {
        public TID winnerThread = null;
        public TID loserThread = null;
        public TID tIDWriter = null;
        public CrystalBox()
        {
            winnerThread = new TID();
            loserThread = new TID();
            tIDWriter = winnerThread;
        }
    }

    class CrystalBoxMuEx
    {
        static int sharedResource = 0;
        static CrystalBox cBox = null;
        public CrystalBoxMuEx()
        {
            cBox = new CrystalBox();
        }

        // This method represents a resource that must be synchronized
        // so that only one thread at a time can enter.
        private static void UseResource()
        {
            int myID = Thread.CurrentThread.ManagedThreadId;
```
// Wait until it is safe to enter.
while (true)
{
    CrystalBox myBox = cBox;
    Console.WriteLine("Thread#{0} is requesting Resource.", myID);
    Thread.VolatileWrite(ref myBox.tIDWriter.thID, myID);
    myBox.tIDWriter = myBox.loserThread;
    //Thread.Sleep(0);
    if (myID == Thread.VolatileRead(ref myBox.winnerThread.thID))
        break;
    Thread.Sleep(1);
}

Console.WriteLine("Thread#{0} has entered the protected area", myID);

// Place code to access non-reentrant resources here.
int local = sharedResource;
// Simulate some work.
Thread.Sleep(5);
sharedResource = local + 1;

Console.WriteLine("Thread#{0} is leaving the protected
t area\r\n SharedResource={1}\r\n", myID, sharedResource);

    // Release the Mutex.
    cBox = new CrystalBox();
    Thread.Sleep(1);
}

/// <summary>
/// The main entry point for the application.
/// </summary>
[STAThread]
static void Main(string[] args)
{
    // Create the threads that will use the protected resource.
    int numThreads = int.Parse(args[0]);
    cBox = new CrystalBox();
    for (int i = 0; i < numThreads; i++)
    {
        Thread myThread = new Thread(new ThreadStart(UseResource));
        myThread.Name = String.Format("Thread{0}" , i + 1);
        myThread.Start();
    }

    // The main thread exits, but the application continues to
    // run until all foreground threads have exited.

}
2. Test case for accessing a shared variable in a loop.

```csharp
using System;
using System.Threading;

namespace CrystalBox
{
    /// <summary>
    /// Summary description for Class1.
    /// </summary>
    class TID
    {
        public int thID = -1;
    }

class CrystalBox
{
    public TID winnerThread = null;
    public TID loserThread = null;
    public TID tIDWriter = null;
    public CrystalBox()
    {
        winnerThread = new TID();
        loserThread = new TID();
        tIDWriter = winnerThread;
    }
}

class CrystalBoxMuEx
{
    static int sharedResource = 0;
    static CrystalBox cBox = null;
    public CrystalBoxMuEx()
    {
        cBox = new CrystalBox();
    }

    private static void MyThreadProc()
    {
        int myID = Thread.CurrentThread.ManagedThreadId;
        for (int i = 0; i < 5; i++)
        {
            Console.WriteLine("Thread#{0} #{1} Time requests Resource.", myID, i);
            UseResource();
        }
    }

    // This method represents a resource that must be synchronized
    // so that only one thread at a time can enter.
    private static void UseResource()
    {
        int myID = Thread.CurrentThread.ManagedThreadId;
        // Wait until it is safe to enter.
        while (true)
        {
            CrystalBox myBox = cBox;
            Console.WriteLine("\tThread#{0} is requesting Resource.", myID);
            Thread.VolatileWrite(ref myBox.tIDWriter.thID, myID);
            myBox.tIDWriter = myBox.loserThread;
            //Thread.Sleep(0);
            if (myID == Thread.VolatileRead(ref myBox.winnerThread.thID))
                break;
            Thread.Sleep(1);
        }
        Console.WriteLine("\tThread#{0} has entered the protected area", myID);
    }
}
```
// Place code to access non-reentrant resources here.
int local=sharedResource;
// Simulate some work.
Thread.Sleep(20);
sharedResource=local+1;

Console.WriteLine("\tThread#{0} is leaving the protected area\n\tSharedResource={1}\n", myID, sharedResource);

// Release the Mutex.
cBox=new CrystalBox();
Thread.Sleep(2);

}/// <summary>
/// The main entry point for the application.
/// </summary>
[STAThread]
static void Main(string[] args)
{
    // Create the threads that will use the protected resource.
    int numThreads=int.Parse(args[0]);
cBox=new CrystalBox();
    for(int i = 0; i < numThreads; i++)
    {
        Thread myThread = new Thread(new ThreadStart(MyThreadProc));
        myThread.Name = String.Format("Thread{0}", i + 1);
        myThread.Start();
    }
    // The main thread exits, but the application continues to run until all foreground threads have exited.
}
3. The new mutual exclusion implementation used for the dining philosophers problem.

```csharp
using System;
using System.Collections;
using System.Threading;

namespace dph
{
    class TID
    {
        public int id;  //Philosopher ID 0-4
    }
    class CrystalBox
    {
        public TID csOwner = new TID();
        public TID IDWriter = null;
        public TID nill = new TID();
        public CrystalBox()
        {
            IDWriter = csOwner;
        }
    }
    class Dph
    {
        // Create 5 crystalBoxs for 5 chopSticks.
        static CrystalBox[] cBox = new CrystalBox[5];
        Dph()
        {
        }

        public void GetChopSticks()
        {
            // left chopStick number
            int i = int.Parse(Thread.CurrentThread.Name);
            // right chopstrick number
            int j = (i + 1) % 5;
            int myID = Thread.CurrentThread.ManagedThreadId;
            for(;;)
            {
                Console.Write(i);
                CrystalBox myboxi = cBox[i];
                Thread.VolatileWrite(ref myboxi.IDWriter.id, myID);
                myboxi.IDWriter = myboxi.nill;
                // check if I have the left chopstick
                if (myID == Thread.VolatileRead(ref myboxi.csOwner.id))
                {
                    CrystalBox myboxj = cBox[j];
                    Thread.VolatileWrite(ref myboxj.IDWriter.id, myID);
                    myboxj.IDWriter = myboxj.nill;
                    // check if can get the right chopstick
                    if (myID == Thread.VolatileRead(ref myboxj.csOwner.id))
                    {
                        break;
                        // got both chopsticks then go to eat
                    }
                    else
                    {
                        cBox[i] = new CrystalBox();  //release left chopstick
                    }
                }
            }
        }
    }
    //Eating
    Console.WriteLine("\nPhilosopher: {0} is eating", i);
    Thread.Sleep(10);
    Console.WriteLine("Philosopher: {0} is done", i);
    cBox[i] = new CrystalBox();
    cBox[j] = new CrystalBox();
}

[STAThread]
static void Main()
{45
```
```csharp
{  
    Console.WriteLine("Table is set");  
    for (int i = 0; i < 5; i++)  
    {
        cBox[i] = new CrystalBox();  
    }

    Thread[] threads = new Thread[5];  
    for (int j = 0; j < 5; j++)  
    {
        Dph d = new Dph();  
        threads[j] = new Thread(new ThreadStart(d.GetChopSticks));  
        threads[j].Name = j.ToString();  
        threads[j].Start();  
    }

    for (int i = 0; i < 5; i++)  
    {
        threads[i].Join();  
    }
}
```
APPENDIX C

RESULTS OF TESTING

This appendix contains three test case results that referenced in Chapter IV. The source codes are listed in Appendix B.

1. The new algorithm used to solve the basic race condition.

   The following result is from using the new algorithm to solve the basic race condition problem. The source code is listed on page 39. Fifty threads were created to increment shared variable by 1. Same program were also run with 100, 200, 300, 400, and 500 threads for a total of 400 times. The result for the case of 50 threads results are listed below.

Thread#4 is requesting Resource. Thread#28 is requesting Resource.
Thread#4 has entered the protected area Thread#29 is requesting Resource.
Thread#5 is requesting Resource. Thread#30 is requesting Resource.
Thread#6 is requesting Resource. Thread#31 is requesting Resource.
Thread#7 is requesting Resource. Thread#32 is requesting Resource.
Thread#8 is requesting Resource. Thread#33 is requesting Resource.
Thread#9 is requesting Resource. Thread#34 is requesting Resource.
Thread#10 is requesting Resource. Thread#35 is requesting Resource.
Thread#11 is requesting Resource. Thread#36 is requesting Resource.
Thread#12 is requesting Resource. Thread#37 is requesting Resource.
Thread#13 is requesting Resource. Thread#38 is requesting Resource.
Thread#14 is requesting Resource. Thread#39 is requesting Resource.
Thread#15 is requesting Resource. Thread#40 is requesting Resource.
Thread#16 is requesting Resource. Thread#41 is requesting Resource.
Thread#17 is requesting Resource. Thread#42 is requesting Resource.
Thread#18 is requesting Resource. Thread#43 is requesting Resource.
Thread#19 is requesting Resource. Thread#44 is requesting Resource.
Thread#20 is requesting Resource. Thread#45 is requesting Resource.
Thread#21 is requesting Resource. Thread#46 is requesting Resource.
Thread#22 is requesting Resource. Thread#47 is requesting Resource.
Thread#23 is requesting Resource. Thread#48 is requesting Resource.
Thread#24 is requesting Resource. Thread#49 is requesting Resource.
Thread#25 is requesting Resource. Thread#50 is requesting Resource.
Thread#26 is requesting Resource. Thread#51 is requesting Resource.
Thread#27 is requesting Resource. Thread#52 is requesting Resource.
Thread#3 is requesting Resource. Thread#10 is requesting Resource.
Thread#9 is requesting Resource.
Thread#13 is requesting Resource.
Thread#17 is requesting Resource.
Thread#21 is requesting Resource.
Thread#25 is requesting Resource.
Thread#29 is requesting Resource.
Thread#33 is requesting Resource.
Thread#7 is requesting Resource.
Thread#6 is requesting Resource.
Thread#14 is requesting Resource.
Thread#8 is requesting Resource.
Thread#12 is requesting Resource.
Thread#11 is requesting Resource.
Thread#15 is requesting Resource.
Thread#19 is requesting Resource.
Thread#23 is requesting Resource.
Thread#27 is requesting Resource.
Thread#31 is requesting Resource.
Thread#35 is requesting Resource.
Thread#39 is requesting Resource.
Thread#43 is requesting Resource.
Thread#37 is requesting Resource.
Thread#41 is requesting Resource.
Thread#45 is requesting Resource.
Thread#49 is requesting Resource.
Thread#48 is requesting Resource.
Thread#44 is requesting Resource.
Thread#40 is requesting Resource.
Thread#36 is requesting Resource.
Thread#20 is requesting Resource.
Thread#16 is requesting Resource.
Thread#29 is requesting Resource.
Thread#25 is requesting Resource.
Thread#12 is requesting Resource.
Thread#21 is requesting Resource.
Thread#8 is requesting Resource.
Thread#18 is requesting Resource.
Thread#32 is requesting Resource.
Thread#36 is requesting Resource.
Thread#40 is requesting Resource.
Thread#34 is requesting Resource.
Thread#10 is requesting Resource.
Thread#12 is requesting Resource.
Thread#8 is requesting Resource.
Thread#32 is requesting Resource.
Thread#28 is requesting Resource.
Thread#24 is leaving the protected area
SharedResource=1

Thread#49 is requesting Resource.
Thread#49 has entered the protected area
Thread#5 is requesting Resource.
Thread#14 is requesting Resource.
Thread#6 is requesting Resource.
Thread#30 is requesting Resource.
Thread#26 is requesting Resource.
Thread#22 is requesting Resource.
Thread#18 is requesting Resource.
Thread#19 is requesting Resource.
Thread#23 is requesting Resource.
Thread#29 is requesting Resource.
Thread#25 is requesting Resource.
Thread#17 is requesting Resource.
Thread#33 is requesting Resource.
Thread#37 is requesting Resource.
Thread#45 is requesting Resource.
Thread#52 is leaving the protected area
SharedResource=2

Thread#4 is leaving the protected area
Thread#52 is leaving the protected area
Thread#31 is requesting Resource.
Thread#35 is requesting Resource.
Thread#39 is requesting Resource.
Thread#43 is requesting Resource.
Thread#37 is requesting Resource.
Thread#41 is requesting Resource.
Thread#47 is requesting Resource.
Thread#51 is requesting Resource.
Thread#45 is requesting Resource.
Thread#18 is requesting Resource.
Thread#22 is requesting Resource.
Thread#26 is requesting Resource.
Thread#30 is requesting Resource.
Thread#34 is requesting Resource.
Thread#38 is requesting Resource.
Thread#42 is requesting Resource.
Thread#46 is requesting Resource.
Thread#50 is requesting Resource.
Thread#24 is requesting Resource.
Thread#28 is requesting Resource.
Thread#32 is requesting Resource.
Thread#36 is requesting Resource.
Thread#40 is requesting Resource.
Thread#44 is requesting Resource.
Thread#48 is requesting Resource.
Thread#4 is leaving the protected area

Thread#49 is requesting Resource.
Thread#49 has entered the protected area
Thread#5 is requesting Resource.
Thread#14 is requesting Resource.
Thread#6 is requesting Resource.
Thread#30 is requesting Resource.
Thread#26 is requesting Resource.
Thread#22 is requesting Resource.
Thread#18 is requesting Resource.
Thread#38 is requesting Resource.
Thread#34 is requesting Resource.
Thread#32 is requesting Resource.
Thread#35 is requesting Resource.
Thread#39 is requesting Resource.
Thread#43 is requesting Resource.
Thread#37 is requesting Resource.
Thread#41 is requesting Resource.
Thread#23 is requesting Resource.
Thread#27 is requesting Resource.
Thread#41 is requesting Resource.
Thread#15 is requesting Resource.
Thread#11 is requesting Resource.
Thread#19 is requesting Resource.
Thread#31 is requesting Resource.
Thread#13 is requesting Resource.
Thread#7 is requesting Resource.
Thread#15 is requesting Resource.
Thread#11 is requesting Resource.
Thread#31 is requesting Resource.
Thread#10 is requesting Resource.
Thread#12 is requesting Resource.
Thread#8 is requesting Resource.
Thread#32 is requesting Resource.
Thread#28 is requesting Resource.
Thread#9 is requesting Resource.
<table>
<thead>
<tr>
<th>Thread#49 is leaving the protected area</th>
<th>Thread#14 is requesting Resource.</th>
</tr>
</thead>
<tbody>
<tr>
<td>SharedResource=3</td>
<td>Thread#9 is requesting Resource.</td>
</tr>
<tr>
<td>Thread#5 is requesting Resource.</td>
<td>Thread#5 is leaving the protected area</td>
</tr>
<tr>
<td>Thread#5 has entered the protected area</td>
<td>SharedResource=4</td>
</tr>
<tr>
<td>Thread#7 is requesting Resource.</td>
<td>Thread#38 is requesting Resource.</td>
</tr>
<tr>
<td>Thread#14 is requesting Resource.</td>
<td>Thread#38 has entered the protected area</td>
</tr>
<tr>
<td>Thread#12 is requesting Resource.</td>
<td>Thread#30 is requesting Resource.</td>
</tr>
<tr>
<td>Thread#24 is requesting Resource.</td>
<td>Thread#26 is requesting Resource.</td>
</tr>
<tr>
<td>Thread#28 is requesting Resource.</td>
<td>Thread#34 is requesting Resource.</td>
</tr>
<tr>
<td>Thread#32 is requesting Resource.</td>
<td>Thread#50 is requesting Resource.</td>
</tr>
<tr>
<td>Thread#8 is requesting Resource.</td>
<td>Thread#18 is requesting Resource.</td>
</tr>
<tr>
<td>Thread#36 is requesting Resource.</td>
<td>Thread#46 is requesting Resource.</td>
</tr>
<tr>
<td>Thread#40 is requesting Resource.</td>
<td>Thread#22 is requesting Resource.</td>
</tr>
<tr>
<td>Thread#44 is requesting Resource.</td>
<td>Thread#6 is requesting Resource.</td>
</tr>
<tr>
<td>Thread#48 is requesting Resource.</td>
<td>Thread#42 is requesting Resource.</td>
</tr>
<tr>
<td>Thread#16 is requesting Resource.</td>
<td>Thread#31 is requesting Resource.</td>
</tr>
<tr>
<td>Thread#20 is requesting Resource.</td>
<td>Thread#8 is requesting Resource.</td>
</tr>
<tr>
<td>Thread#3 is requesting Resource.</td>
<td>Thread#23 is requesting Resource.</td>
</tr>
<tr>
<td>Thread#7 is requesting Resource.</td>
<td>Thread#27 is requesting Resource.</td>
</tr>
<tr>
<td>Thread#26 is requesting Resource.</td>
<td>Thread#47 is requesting Resource.</td>
</tr>
<tr>
<td>Thread#30 is requesting Resource.</td>
<td>Thread#39 is requesting Resource.</td>
</tr>
<tr>
<td>Thread#38 is requesting Resource.</td>
<td>Thread#43 is requesting Resource.</td>
</tr>
<tr>
<td>Thread#42 is requesting Resource.</td>
<td>Thread#15 is requesting Resource.</td>
</tr>
<tr>
<td>Thread#46 is requesting Resource.</td>
<td>Thread#3 is requesting Resource.</td>
</tr>
<tr>
<td>Thread#18 is requesting Resource.</td>
<td>Thread#50 is requesting Resource.</td>
</tr>
<tr>
<td>Thread#50 is requesting Resource.</td>
<td>Thread#7 is requesting Resource.</td>
</tr>
<tr>
<td>Thread#34 is requesting Resource.</td>
<td>Thread#11 is requesting Resource.</td>
</tr>
<tr>
<td>Thread#6 is requesting Resource.</td>
<td>Thread#32 is requesting Resource.</td>
</tr>
<tr>
<td>Thread#22 is requesting Resource.</td>
<td>Thread#44 is requesting Resource.</td>
</tr>
<tr>
<td>Thread#10 is requesting Resource.</td>
<td>Thread#12 is requesting Resource.</td>
</tr>
<tr>
<td>Thread#19 is requesting Resource.</td>
<td>Thread#24 is requesting Resource.</td>
</tr>
<tr>
<td>Thread#25 is requesting Resource.</td>
<td>Thread#28 is requesting Resource.</td>
</tr>
<tr>
<td>Thread#31 is requesting Resource.</td>
<td>Thread#40 is requesting Resource.</td>
</tr>
<tr>
<td>Thread#23 is requesting Resource.</td>
<td>Thread#36 is requesting Resource.</td>
</tr>
<tr>
<td>Thread#43 is requesting Resource.</td>
<td>Thread#40 is requesting Resource.</td>
</tr>
<tr>
<td>Thread#39 is requesting Resource.</td>
<td>Thread#48 is requesting Resource.</td>
</tr>
<tr>
<td>Thread#47 is requesting Resource.</td>
<td>Thread#35 is requesting Resource.</td>
</tr>
<tr>
<td>Thread#27 is requesting Resource.</td>
<td>Thread#29 is requesting Resource.</td>
</tr>
<tr>
<td>Thread#7 is requesting Resource.</td>
<td>Thread#41 is requesting Resource.</td>
</tr>
<tr>
<td>Thread#3 is requesting Resource.</td>
<td>Thread#8 is requesting Resource.</td>
</tr>
<tr>
<td>Thread#15 is requesting Resource.</td>
<td>Thread#13 is requesting Resource.</td>
</tr>
<tr>
<td>Thread#32 is requesting Resource.</td>
<td>Thread#37 is requesting Resource.</td>
</tr>
<tr>
<td>Thread#28 is requesting Resource.</td>
<td>Thread#41 is requesting Resource.</td>
</tr>
<tr>
<td>Thread#24 is requesting Resource.</td>
<td>Thread#25 is requesting Resource.</td>
</tr>
<tr>
<td>Thread#12 is requesting Resource.</td>
<td>Thread#33 is requesting Resource.</td>
</tr>
<tr>
<td>Thread#44 is requesting Resource.</td>
<td>Thread#21 is requesting Resource.</td>
</tr>
<tr>
<td>Thread#10 is requesting Resource.</td>
<td>Thread#40 is requesting Resource.</td>
</tr>
<tr>
<td>Thread#40 is requesting Resource.</td>
<td>Thread#36 is requesting Resource.</td>
</tr>
<tr>
<td>Thread#20 is requesting Resource.</td>
<td>Thread#17 is requesting Resource.</td>
</tr>
<tr>
<td>Thread#16 is requesting Resource.</td>
<td>Thread#17 is requesting Resource.</td>
</tr>
<tr>
<td>Thread#48 is requesting Resource.</td>
<td>Thread#39 is requesting Resource.</td>
</tr>
<tr>
<td>Thread#13 is requesting Resource.</td>
<td>Thread#37 is requesting Resource.</td>
</tr>
<tr>
<td>Thread#29 is requesting Resource.</td>
<td>Thread#14 is requesting Resource.</td>
</tr>
<tr>
<td>Thread#25 is requesting Resource.</td>
<td>Thread#50 is requesting Resource.</td>
</tr>
<tr>
<td>Thread#33 is requesting Resource.</td>
<td>Thread#34 is requesting Resource.</td>
</tr>
<tr>
<td>Thread#37 is requesting Resource.</td>
<td>Thread#26 is requesting Resource.</td>
</tr>
<tr>
<td>Thread#41 is requesting Resource.</td>
<td>Thread#42 is requesting Resource.</td>
</tr>
<tr>
<td>Thread#45 is requesting Resource.</td>
<td>Thread#6 is requesting Resource.</td>
</tr>
<tr>
<td>Thread#21 is requesting Resource.</td>
<td>Thread#22 is requesting Resource.</td>
</tr>
<tr>
<td>Thread#17 is requesting Resource.</td>
<td>Thread#46 is requesting Resource.</td>
</tr>
<tr>
<td>Thread#11 is requesting Resource.</td>
<td>Thread#38 is leaving the protected area</td>
</tr>
</tbody>
</table>
SharedResource=5
Thread#18 is requesting Resource.
Thread#19 is requesting Resource.
Thread#10 is requesting Resource.
Thread#39 is requesting Resource.
Thread#39 has entered the protected area
Thread#27 is requesting Resource.
Thread#13 is requesting Resource.
Thread#7 is requesting Resource.
Thread#3 is requesting Resource.
Thread#29 is requesting Resource.
Thread#33 is requesting Resource.
Thread#15 is requesting Resource.
Thread#43 is requesting Resource.
Thread#25 is requesting Resource.
Thread#41 is requesting Resource.
Thread#11 is requesting Resource.
Thread#37 is requesting Resource.
Thread#45 is requesting Resource.
Thread#35 is requesting Resource.
Thread#51 is requesting Resource.
Thread#9 is requesting Resource.
Thread#17 is requesting Resource.
Thread#44 is requesting Resource.
Thread#21 is requesting Resource.
Thread#23 is requesting Resource.
Thread#16 is requesting Resource.
Thread#14 is requesting Resource.
Thread#22 is requesting Resource.
Thread#29 is requesting Resource.
Thread#18 is requesting Resource.
Thread#10 is requesting Resource.
Thread#27 is requesting Resource.
Thread#13 is requesting Resource.
Thread#33 is requesting Resource.
Thread#29 is requesting Resource.
Thread#41 is requesting Resource.
Thread#25 is requesting Resource.
Thread#45 is requesting Resource.
Thread#37 is requesting Resource.
Thread#39 is requesting Resource.
Thread#31 is requesting Resource.
Thread#5 is requesting Resource.
Thread#8 is requesting Resource.
Thread#36 is requesting Resource.
Thread#42 is requesting Resource.
Thread#26 is requesting Resource.
Thread#34 is requesting Resource.
Thread#18 is requesting Resource.
Thread#47 is requesting Resource.
Thread#46 is requesting Resource.
Thread#23 is requesting Resource.
Thread#10 is requesting Resource.
Thread#3 is requesting Resource.
Thread#10 is requesting Resource.
Thread#27 is requesting Resource.
Thread#13 is requesting Resource.
Thread#33 is requesting Resource.
Thread#29 is requesting Resource.
Thread#41 is requesting Resource.
Thread#25 is requesting Resource.
Thread#45 is requesting Resource.
Thread#37 is requesting Resource.
Thread#39 is requesting Resource.
Thread#9 is requesting Resource.
Thread#21 is requesting Resource.
Thread#50 is requesting Resource.
Thread#42 is requesting Resource.
Thread#6 is requesting Resource.
Thread#30 is requesting Resource.
Thread#14 is requesting Resource.
Thread#18 is requesting Resource.
Thread#34 is requesting Resource.
Thread#10 is requesting Resource.
Thread#27 is requesting Resource.
Thread#13 is requesting Resource.
Thread#33 is requesting Resource.
Thread#29 is requesting Resource.
Thread#41 is requesting Resource.
Thread#25 is requesting Resource.
Thread#45 is requesting Resource.
Thread#37 is requesting Resource.
Thread#39 is requesting Resource.
Thread#17 is requesting Resource.
Thread#9 is requesting Resource.

Thread#21 is requesting Resource.
Thread#50 is requesting Resource.
Thread#42 is requesting Resource.
Thread#6 is requesting Resource.
Thread#30 is requesting Resource.
Thread#14 is requesting Resource.
Thread#18 is requesting Resource.
Thread#34 is requesting Resource.
Thread#10 is requesting Resource.
Thread#27 is requesting Resource.
Thread#13 is requesting Resource.
Thread#33 is requesting Resource.
Thread#29 is requesting Resource.
Thread#41 is requesting Resource.
Thread#25 is requesting Resource.
Thread#45 is requesting Resource.
Thread#37 is requesting Resource.
Thread#39 is requesting Resource.
Thread#17 is requesting Resource.
Thread#9 is requesting Resource.
Thread#21 is requesting Resource.
Thread#50 is requesting Resource.
Thread#42 is requesting Resource.
Thread#6 is requesting Resource.
Thread#30 is requesting Resource.
Thread#14 is requesting Resource.
Thread#18 is requesting Resource.
Thread#34 is requesting Resource.
Thread#10 is requesting Resource.
Thread#27 is requesting Resource.
Thread#13 is requesting Resource.
Thread#33 is requesting Resource.
Thread#29 is requesting Resource.
Thread#41 is requesting Resource.
Thread#25 is requesting Resource.
Thread#45 is requesting Resource.
Thread#37 is requesting Resource.
Thread#39 is requesting Resource.
Thread#17 is requesting Resource.
Thread#9 is requesting Resource.
Thread#21 is requesting Resource.
Thread#50 is requesting Resource.
Thread#42 is requesting Resource.
Thread#6 is requesting Resource.
Thread#30 is requesting Resource.
Thread#14 is requesting Resource.
Thread#18 is requesting Resource.
Thread#34 is requesting Resource.
Thread#10 is requesting Resource.
Thread#27 is requesting Resource.
Thread#13 is requesting Resource.
Thread#33 is requesting Resource.
Thread#29 is requesting Resource.
Thread#41 is requesting Resource.
Thread#25 is requesting Resource.
Thread#45 is requesting Resource.
Thread#37 is requesting Resource.
Thread#39 is requesting Resource.
Thread#17 is requesting Resource.
Thread#9 is requesting Resource.
Thread#21 is requesting Resource.
Thread#50 is requesting Resource.
Thread#42 is requesting Resource.
Thread#6 is requesting Resource.
Thread#30 is requesting Resource.
Thread#14 is requesting Resource.
Thread#18 is requesting Resource.
Thread#34 is requesting Resource.
Thread#10 is requesting Resource.
Thread#27 is requesting Resource.
Thread#13 is requesting Resource.
Thread#33 is requesting Resource.
Thread#47 is requesting Resource.
Thread#19 is requesting Resource.
Thread#7 is requesting Resource.
Thread#43 is requesting Resource.
Thread#48 is requesting Resource.
Thread#40 is requesting Resource.
Thread#32 is requesting Resource.
Thread#20 is requesting Resource.
Thread#8 is requesting Resource.
Thread#16 is requesting Resource.
Thread#24 is requesting Resource.
Thread#12 is requesting Resource.
Thread#44 is requesting Resource.
Thread#36 is requesting Resource.
Thread#28 is leaving the protected area
SharedResource=7
Thread#3 is requesting Resource.
Thread#10 is requesting Resource.
Thread#51 is requesting Resource.
Thread#15 is requesting Resource.
Thread#15 has entered the protected area
Thread#35 is requesting Resource.
Thread#31 is requesting Resource.
Thread#19 is requesting Resource.
Thread#47 is requesting Resource.
Thread#47 is requesting Resource.
Thread#23 is requesting Resource.
Thread#11 is requesting Resource.
Thread#43 is requesting Resource.
Thread#10 is requesting Resource.
Thread#7 is requesting Resource.
Thread#51 is requesting Resource.
Thread#3 is requesting Resource.
Thread#40 is requesting Resource.
Thread#48 is requesting Resource.
Thread#16 is requesting Resource.
Thread#8 is requesting Resource.
Thread#16 is requesting Resource.
Thread#40 is requesting Resource.
Thread#10 is requesting Resource.
Thread#43 is requesting Resource.
Thread#23 is requesting Resource.
Thread#35 is requesting Resource.
Thread#31 is requesting Resource.
Thread#19 is requesting Resource.
Thread#51 is requesting Resource.
Thread#3 is requesting Resource.
Thread#25 is requesting Resource.
Thread#19 is requesting Resource.
Thread#6 is requesting Resource.
Thread#42 is requesting Resource.
Thread#50 is requesting Resource.
Thread#34 is requesting Resource.
Thread#18 is requesting Resource.
Thread#14 is requesting Resource.
Thread#10 is requesting Resource.
Thread#16 is requesting Resource.
Thread#40 is requesting Resource.
Thread#12 is requesting Resource.
Thread#44 is requesting Resource.
Thread#36 is requesting Resource.
Thread#10 is requesting Resource.
Thread#10 is requesting Resource.
Thread#11 is requesting Resource.
Thread#37 is requesting Resource.
Thread#37 is requesting Resource.
Thread#21 is requesting Resource.
Thread#27 is requesting Resource.
Thread#13 is requesting Resource.
Thread#15 is leaving the protected area
SharedResource=8
Thread#20 is requesting Resource.
Thread#20 has entered the protected area
Thread#8 is requesting Resource.
Thread#46 is requesting Resource.
Thread#46 is requesting Resource.
Thread#32 is requesting Resource.
Thread#24 is requesting Resource.
Thread#42 is requesting Resource.
Thread#6 is requesting Resource.
Thread#30 is requesting Resource.
Thread#14 is requesting Resource.
Thread#18 is requesting Resource.
Thread#18 is requesting Resource.
Thread#21 is requesting Resource.
Thread#27 is requesting Resource.
Thread#13 is requesting Resource.
Thread#15 is leaving the protected area
SharedResource=8
Thread#20 is requesting Resource.
Thread#20 has entered the protected area
Thread#8 is requesting Resource.
Thread#46 is requesting Resource.
Thread#46 is requesting Resource.
Thread#32 is requesting Resource.
Thread#24 is requesting Resource.
Thread#42 is requesting Resource.
Thread#6 is requesting Resource.
Thread#33 is requesting Resource. Thread#31 is requesting Resource.
Thread#29 is requesting Resource. Thread#31 has entered the protected area
Thread#41 is requesting Resource. Thread#41 is requesting Resource.
Thread#21 is requesting Resource. Thread#21 is requesting Resource.
Thread#45 is requesting Resource. Thread#25 is requesting Resource.
Thread#37 is requesting Resource. Thread#17 is requesting Resource.
Thread#17 is requesting Resource. Thread#13 is requesting Resource.
Thread#25 is requesting Resource. Thread#45 is requesting Resource.
Thread#25 is requesting Resource. Thread#13 is requesting Resource.
Thread#13 is requesting Resource. Thread#45 is requesting Resource.
Thread#6 is requesting Resource. Thread#19 is requesting Resource.
Thread#18 is requesting Resource. Thread#51 is requesting Resource.
Thread#34 is requesting Resource. Thread#11 is requesting Resource.
Thread#50 is requesting Resource. Thread#23 is requesting Resource.
Thread#50 is requesting Resource. Thread#43 is requesting Resource.
Thread#42 is requesting Resource. Thread#27 is requesting Resource.
Thread#42 is requesting Resource. Thread#27 is requesting Resource.
Thread#22 is requesting Resource. Thread#27 is requesting Resource.
Thread#46 is requesting Resource. Thread#27 is requesting Resource.
Thread#46 is requesting Resource. Thread#47 is requesting Resource.
Thread#30 is requesting Resource. Thread#47 is requesting Resource.
Thread#30 is requesting Resource. Thread#35 is requesting Resource.
Thread#30 is requesting Resource. Thread#35 is requesting Resource.
Thread#14 is requesting Resource. Thread#7 is requesting Resource.
Thread#14 is requesting Resource. Thread#31 is requesting Resource.
Thread#26 is requesting Resource. Thread#19 is requesting Resource.
Thread#26 is requesting Resource. Thread#3 is leaving the protected area
Thread#16 is requesting Resource. Thread#3 is leaving the protected area
Thread#40 is requesting Resource. Thread#31 is requesting Resource.
Thread#34 is requesting Resource. SharedResource=10
Thread#40 is requesting Resource. Thread#40 is requesting Resource.
Thread#40 is requesting Resource. Thread#42 is requesting Resource.
Thread#40 is requesting Resource. Thread#42 is requesting Resource.
Thread#40 is requesting Resource. Thread#42 is requesting Resource.
Thread#40 is requesting Resource. Thread#42 is requesting Resource.
Thread#40 is requesting Resource. Thread#42 is requesting Resource.
Thread#40 is requesting Resource. Thread#42 is requesting Resource.
Thread#40 is requesting Resource. Thread#42 is requesting Resource.
Thread#40 is requesting Resource. Thread#42 is requesting Resource.
Thread#40 is requesting Resource. Thread#42 is requesting Resource.
Thread#40 is requesting Resource. Thread#42 is requesting Resource.
Thread#40 is requesting Resource. Thread#42 is requesting Resource.
Thread#40 is requesting Resource. Thread#42 is requesting Resource.
Thread#40 is requesting Resource. Thread#42 is requesting Resource.
Thread#40 is requesting Resource. Thread#42 is requesting Resource.
Thread#40 is requesting Resource. Thread#42 is requesting Resource.
Thread#40 is requesting Resource. Thread#42 is requesting Resource.
Thread#40 is requesting Resource. Thread#42 is requesting Resource.
Thread#40 is requesting Resource. Thread#42 is requesting Resource.
Thread#40 is requesting Resource. Thread#42 is requesting Resource.
Thread#40 is requesting Resource. Thread#42 is requesting Resource.
Thread#40 is requesting Resource. Thread#42 is requesting Resource.
Thread#40 is requesting Resource. Thread#42 is requesting Resource.
Thread#40 is requesting Resource. Thread#42 is requesting Resource.
Thread#40 is requesting Resource. Thread#42 is requesting Resource.
Thread#40 is requesting Resource. Thread#42 is requesting Resource.
Thread#40 is requesting Resource. Thread#42 is requesting Resource.
Thread#40 is requesting Resource. Thread#42 is requesting Resource.
Thread#40 is requesting Resource. Thread#42 is requesting Resource.
Thread#40 is requesting Resource. Thread#42 is requesting Resource.
Thread#40 is requesting Resource. Thread#42 is requesting Resource.
Thread#40 is requesting Resource. Thread#42 is requesting Resource.
Thread#40 is requesting Resource. Thread#42 is requesting Resource.
Thread#40 is requesting Resource. Thread#42 is requesting Resource.
Thread#40 is requesting Resource. Thread#42 is requesting Resource.
Thread#40 is requesting Resource. Thread#42 is requesting Resource.
Thread#40 is requesting Resource. Thread#42 is requesting Resource.
Thread#40 is requesting Resource. Thread#42 is requesting Resource.
Thread#40 is requesting Resource. Thread#42 is requesting Resource.
Thread#40 is requesting Resource. Thread#42 is requesting Resource.
Thread#40 is requesting Resource. Thread#42 is requesting Resource.
Thread#40 is requesting Resource. Thread#42 is requesting Resource.
Thread#40 is requesting Resource. Thread#42 is requesting Resource.
Thread#40 is requesting Resource. Thread#42 is requesting Resource.
Thread#40 is requesting Resource. Thread#42 is requesting Resource.
Thread#40 is requesting Resource. Thread#42 is requesting Resource.
Thread#40 is requesting Resource. Thread#42 is requesting Resource.
Thread#40 is requesting Resource. Thread#42 is requesting Resource.
Thread#40 is requesting Resource. Thread#42 is requesting Resource.
Thread#40 is requesting Resource. Thread#42 is requesting Resource.
Thread#40 is requesting Resource. Thread#42 is requesting Resource.
Thread#40 is requesting Resource. Thread#42 is requesting Resource.
Thread#40 is requesting Resource. Thread#42 is requesting Resource.
Thread#40 is requesting Resource. Thread#42 is requesting Resource.
Thread#40 is requesting Resource. Thread#42 is requesting Resource.
Thread#40 is requesting Resource. Thread#42 is requesting Resource.
Thread#40 is requesting Resource. Thread#42 is requesting Resource.
Thread#40 is requesting Resource. Thread#42 is requesting Resource.
Thread#40 is requesting Resource. Thread#42 is requesting Resource.
Thread#40 is requesting Resource. Thread#42 is requesting Resource.
Thread#40 is requesting Resource. Thread#42 is requesting Resource.
Thread#40 is requesting Resource. Thread#42 is requesting Resource.
Thread#40 is requesting Resource. Thread#42 is requesting Resource.
Thread#40 is requesting Resource. Thread#42 is requesting Resource.
Thread#40 is requesting Resource. Thread#42 is requesting Resource.
Thread#40 is requesting Resource. Thread#42 is requesting Resource.
Thread#40 is requesting Resource. Thread#42 is requesting Resource.
Thread#40 is requesting Resource. Thread#42 is requesting Resource.
Thread#40 is requesting Resource. Thread#42 is requesting Resource.
Thread#40 is requesting Resource. Thread#42 is requesting Resource.
Thread#40 is requesting Resource. Thread#42 is requesting Resource.
Thread#40 is requesting Resource. Thread#42 is requesting Resource.
Thread#40 is requesting Resource. Thread#42 is requesting Resource.
Thread#40 is requesting Resource. Thread#42 is requesting Resource.
Thread#40 is requesting Resource. Thread#42 is requesting Resource.
Thread#40 is requesting Resource. Thread#42 is requesting Resource.
Thread#40 is requesting Resource. Thread#42 is requesting Resource.
Thread#40 is requesting Resource. Thread#42 is requesting Resource.
Thread#40 is requesting Resource. Thread#42 is requesting Resource.
Thread#40 is requesting Resource. Thread#42 is requesting Resource.
Thread#40 is requesting Resource. Thread#42 is requesting Resource.
Thread#40 is requesting Resource. Thread#42 is requesting Resource.
Thread#40 is requesting Resource. Thread#42 is requesting Resource.
Thread#40 is requesting Resource. Thread#42 is requesting Resource.
Thread#40 is requesting Resource. Thread#42 is requesting Resource.
Thread#40 is requesting Resource. Thread#42 is requesting Resource.
Thread#40 is requesting Resource. Thread#42 is requesting Resource.
Thread#40 is requesting Resource. Thread#42 is requesting Resource.
Thread#40 is requesting Resource. Thread#42 is requesting Resource.
Thread#40 is requesting Resource. Thread#42 is requesting Resource.
Thread#40 is requesting Resource. Thread#42 is requesting Resource.
Thread#40 is requesting Resource. Thread#42 is requesting Resource.
Thread#40 is requesting Resource. Thread#42 is requesting Resource.
Thread#40 is requesting Resource. Thread#42 is requesting Resource.
Thread#40 is requesting Resource. Thread#42 is requesting Resource.
Thread#40 is requesting Resource. Thread#42 is requesting Resource.
Thread#40 is requesting Resource. Thread#42 is requesting Resource.
Thread#40 is requesting Resource. Thread#42 is requesting Resource.
Thread#40 is requesting Resource. Thread#42 is requesting Resource.
Thread#40 is requesting Resource. Thread#42 is requesting Resource.
Thread#40 is requesting Resource. Thread#42 is requesting Resource.
Thread#40 is requesting Resource. Thread#42 is requesting Resource.
Thread#40 is requesting Resource. Thread#42 is requesting Resource.
Thread#40 is requesting Resource. Thread#42 is requesting Resource.
Thread#40 is requesting Resource. Thread#42 is requesting Resource.
Thread#40 is requesting Resource. Thread#42 is requesting Resource.
Thread#40 is requesting Resource. Thread#42 is requesting Resource.
Thread#40 is requesting Resource. Thread#42 is requesting Resource.
Thread#40 is requesting Resource. Thread#42 is requesting Resource.
Thread#40 is requesting Resource. Thread#42 is requesting Resource.
Thread#40 is requesting Resource. Thread#42 is requesting Resource.
Thread#40 is requesting Resource. Thread#42 is requesting Resource.
Thread#40 is requesting Resource. Thread#42 is requesting Resource.
Thread#40 is requesting Resource. Thread#42 is requesting Resource.
Thread#40 is requesting Resource. Thread#42 is requesting Resource.
Thread#40 is requesting Resource. Thread#42 is requesting Resource.
Thread#40 is requesting Resource. Thread#42 is requesting Resource.
Thread#40 is requesting Resource. Thread#42 is requesting Resource.
Thread#40 is requesting Resource. Thread#42 is requesting Resource.
Thread#40 is requesting Resource. Thread#42 is requesting Resource.
Thread#40 is requesting Resource. Thread#42 is requesting Resource.
Thread#40 is requesting Resource. Thread#42 is requesting Resource.
Thread#40 is requesting Resource. Thread#42 is requesting Resource.
Thread#40 is requesting Resource. Thread#42 is requesting Resource.
Thread#40 is requesting Resource. Thread#42 is requesting Resource.
Thread#40 is requesting Resource. Thread#42 is requesting Resource.
Thread#40 is requesting Resource. Thread#42 is requesting Resource.
Thread#40 is requesting Resource. Thread#42 is requesting Resource.
Thread#48 is requesting Resource.
Thread#23 is requesting Resource.
Thread#34 is requesting Resource.
Thread#18 is requesting Resource.
Thread#14 is requesting Resource.
Thread#46 is requesting Resource.
Thread#19 is requesting Resource.
Thread#31 is requesting Resource.
Thread#22 is requesting Resource.
Thread#26 is requesting Resource.
Thread#12 is requesting Resource.
Thread#24 is requesting Resource.
Thread#50 is requesting Resource.
Thread#6 is requesting Resource.
Thread#16 is requesting Resource.
Thread#36 is requesting Resource.
Thread#42 is requesting Resource.
Thread#30 is requesting Resource.
Thread#6 is requesting Resource.
Thread#37 is requesting Resource.
Thread#44 is leaving the protected area
SharedResource=11

Thread#29 is requesting Resource.
Thread#29 has entered the protected area
Thread#33 is requesting Resource.
Thread#9 is requesting Resource.
Thread#13 is requesting Resource.
Thread#45 is requesting Resource.
Thread#41 is requesting Resource.
Thread#21 is requesting Resource.
Thread#17 is requesting Resource.
2. Test result for 10 threads update shared variable 5 times.

The following test result shows 10 threads updating a shared variable 5 times. Each thread tries to update the shared variable in a loop. The purpose of this test is to demonstrate that each thread will yield access to other threads after each critical section completion; therefore progress is assured.

<table>
<thead>
<tr>
<th>Thread#3</th>
<th>#0 Time requests Resource.</th>
<th>Thread#7</th>
<th>is requesting Resource.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread#4</td>
<td>#0 Time requests Resource.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Thread#5</td>
<td>#0 Time requests Resource.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Thread#6</td>
<td>#0 Time requests Resource.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Thread#3</td>
<td>is requesting Resource.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Thread#3</td>
<td>has entered the protected area</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Thread#4</td>
<td>is requesting Resource.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Thread#5</td>
<td>is requesting Resource.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Thread#6</td>
<td>is requesting Resource.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Thread#7</td>
<td>#0 Time requests Resource.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Thread#7</td>
<td>is requesting Resource.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Thread#6</td>
<td>is requesting Resource.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Thread#8</td>
<td>#0 Time requests Resource.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Thread#8</td>
<td>is requesting Resource.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Thread#9</td>
<td>#0 Time requests Resource.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Thread#9</td>
<td>is requesting Resource.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Thread#10</td>
<td>#0 Time requests Resource.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Thread#10</td>
<td>is requesting Resource.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Thread#11</td>
<td>#0 Time requests Resource.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Thread#11</td>
<td>is requesting Resource.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Thread#12</td>
<td>#0 Time requests Resource.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Thread#12</td>
<td>is requesting Resource.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Thread#12</td>
<td>is requesting Resource.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Thread#7</td>
<td>is requesting Resource.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Thread#7</td>
<td>has entered the protected area</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Thread#4</td>
<td>is requesting Resource.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Thread#10</td>
<td>is requesting Resource.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Thread#6</td>
<td>is requesting Resource.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Thread#5</td>
<td>is requesting Resource.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Thread#5</td>
<td>is requesting Resource.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Thread#5</td>
<td>is requesting Resource.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Thread#4</td>
<td>is requesting Resource.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Thread#3</td>
<td>is requesting Resource.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Thread#11</td>
<td>is requesting Resource.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Thread#7</td>
<td>is requesting Resource.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Thread#9</td>
<td>is requesting Resource.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Thread#8</td>
<td>is requesting Resource.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Thread#12</td>
<td>is requesting Resource.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Thread#11</td>
<td>is requesting Resource.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Thread#3</td>
<td>is requesting Resource.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Thread#10</td>
<td>is requesting Resource.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Thread#6</td>
<td>is requesting Resource.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Thread#4</td>
<td>is requesting Resource.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Thread#5</td>
<td>#1 Time requests Resource.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Thread#5</td>
<td>is requesting Resource.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Thread#4</td>
<td>is requesting Resource.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Thread#12</td>
<td>is requesting Resource.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Thread#5</td>
<td>is requesting Resource.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Thread#10</td>
<td>is requesting Resource.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Thread#6</td>
<td>is requesting Resource.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Thread#3</td>
<td>is requesting Resource.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Thread#9</td>
<td>is requesting Resource.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Thread#7</td>
<td>is requesting Resource.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Thread#9</td>
<td>has entered the protected area</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Thread#6</td>
<td>is requesting Resource.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Thread#10</td>
<td>is requesting Resource.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Thread#7</td>
<td>is requesting Resource.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Thread#4</td>
<td>is requesting Resource.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Thread#12</td>
<td>is requesting Resource.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Thread#5</td>
<td>#1 Time requests Resource.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Thread#5</td>
<td>is requesting Resource.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Thread#4</td>
<td>is requesting Resource.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Thread#12</td>
<td>is requesting Resource.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Thread#5</td>
<td>is requesting Resource.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Thread#10</td>
<td>is requesting Resource.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Thread#6</td>
<td>is requesting Resource.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Thread#3</td>
<td>is requesting Resource.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Thread#9</td>
<td>is requesting Resource.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Thread#7</td>
<td>is requesting Resource.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Thread#8</td>
<td>is requesting Resource.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Thread#5</td>
<td>is leaving the protected area</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SharedResource=1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Thread#7</td>
<td>is requesting Resource.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Thread#5</td>
<td>is requesting Resource.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Thread#5</td>
<td>has entered the protected area</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Thread#6</td>
<td>is requesting Resource.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Thread#9</td>
<td>is requesting Resource.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Thread#4</td>
<td>is requesting Resource.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Thread#10</td>
<td>is requesting Resource.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Thread#10 has entered the protected area
Thread#6 is requesting Resource.
Thread#4 is requesting Resource.
Thread#7 is requesting Resource.
Thread#12 is requesting Resource.
Thread#8 is requesting Resource.
Thread#9 is requesting Resource.
Thread#11 #1 Time requests Resource.
Thread#9 is requesting Resource.
Thread#11 is requesting Resource.
Thread#5 is requesting Resource.
Thread#3 is requesting Resource.
Thread#7 is requesting Resource.
Thread#3 is requesting Resource.
Thread#12 is requesting Resource.
Thread#8 is requesting Resource.
Thread#9 is requesting Resource.
Thread#11 is requesting Resource.
Thread#5 is requesting Resource.
Thread#10 #1 Time requests Resource.
Thread#11 is requesting Resource.
Thread#9 is requesting Resource.
Thread#11 is requesting Resource.
Thread#6 is requesting Resource.
Thread#8 is requesting Resource.
Thread#5 is requesting Resource.
Thread#10 is requesting Resource.
Thread#4 is requesting Resource.
Thread#9 is requesting Resource.
Thread#7 is requesting Resource.
Thread#3 is requesting Resource.
Thread#12 is requesting Resource.
Thread#8 is requesting Resource.
Thread#9 is requesting Resource.
Thread#11 is requesting Resource.
Thread#5 is requesting Resource.
Thread#12 has entered the protected area
SharedResource=4
Thread#6 is requesting Resource.
Thread#6 has entered the protected area
Thread#3 is requesting Resource.
Thread#7 is requesting Resource.
Thread#4 is requesting Resource.
Thread#11 is requesting Resource.
Thread#3 is requesting Resource.
Thread#12 is requesting Resource.
Thread#8 is requesting Resource.
Thread#5 is requesting Resource.
Thread#10 #1 Time requests Resource.
Thread#10 is requesting Resource.
Thread#11 is requesting Resource.
Thread#9 is requesting Resource.
Thread#7 is requesting Resource.
Thread#8 is requesting Resource.
Thread#4 is requesting Resource.
Thread#12 is requesting Resource.
Thread#10 is requesting Resource.
Thread#6 is leaving the protected area
SharedResource=5
Thread#5 is requesting Resource.
Thread#12 is requesting Resource.
Thread#12 has entered the protected area
Thread#7 is requesting Resource.
Thread#11 is requesting Resource.
Thread#11 has entered the protected area
Thread#3 is requesting Resource.
Thread#7 is requesting Resource.
Thread#11 is requesting Resource.
Thread#12 is requesting Resource.
Thread#8 is requesting Resource.
Thread#5 is requesting Resource.
Thread#10 #1 Time requests Resource.
Thread#10 is requesting Resource.
Thread#3 is requesting Resource.
Thread#7 is requesting Resource.
Thread#8 is requesting Resource.
Thread#4 is requesting Resource.
Thread#12 is requesting Resource.
Thread#9 is requesting Resource.
Thread#7 is requesting Resource.
Thread#8 is requesting Resource.
Thread#4 is requesting Resource.
Thread#10 is requesting Resource.
Thread#11 is requesting Resource.
Thread#6 is leaving the protected area
SharedResource=6
Thread#5 is requesting Resource.
Thread#11 is requesting Resource.
Thread#11 has entered the protected area
Thread#3 is requesting Resource.
Thread#7 is requesting Resource.
Thread#12 #1 Time requests Resource.
Thread#12 is requesting Resource.
Thread#12 is requesting Resource.
Thread#10 #1 Time requests Resource.
Thread#10 is requesting Resource.
Thread#3 is requesting Resource.
Thread#7 is requesting Resource.
Thread#8 is requesting Resource.
Thread#4 is requesting Resource.
Thread#12 is requesting Resource.
Thread#9 is requesting Resource.
Thread#7 is requesting Resource.
Thread#8 is requesting Resource.
Thread#4 is requesting Resource.
Thread#10 is requesting Resource.
Thread#11 is requesting Resource.
Thread#6 is leaving the protected area
SharedResource=7
Thread#5 is requesting Resource.
Thread#11 is requesting Resource.
Thread#11 has entered the protected area
Thread#3 is requesting Resource.
Thread#7 is requesting Resource.
Thread#4 is requesting Resource.
Thread#5 is requesting Resource.
Thread#12 is requesting Resource.
Thread#9 is requesting Resource.
Thread#8 is requesting Resource.
Thread#8 is requesting Resource.
Thread#7 is requesting Resource.
Thread#7 is requesting Resource.
Thread#7 is requesting Resource.
Thread#5 is requesting Resource.
Thread#5 is requesting Resource.
Thread#5 is requesting Resource.
Thread#5 is requesting Resource.
Thread#6 #2 Time requests Resource.
Thread#6 is requesting Resource.
Thread#3 is requesting Resource.
Thread#10 is requesting Resource.
Thread#4 is requesting Resource.
Thread#8 is requesting Resource.
Thread#8 is requesting Resource.
Thread#3 is requesting Resource.
Thread#11 #1 Time requests Resource.
Thread#11 is requesting Resource.
Thread#4 is requesting Resource.
Thread#4 has entered the
protected area
Thread#5 is requesting Resource.
Thread#10 is requesting Resource.
Thread#9 is requesting Resource.
Thread#6 is requesting Resource.
Thread#7 is requesting Resource.
Thread#7 is requesting Resource.
Thread#3 is requesting Resource.
Thread#5 is requesting Resource.
Thread#5 is requesting Resource.
Thread#6 is requesting Resource.
Thread#6 is requesting Resource.
Thread#6 is requesting Resource.
Thread#5 is requesting Resource.
Thread#9 is requesting Resource.
Thread#10 is requesting Resource.
Thread#12 is requesting Resource.
Thread#11 is leaving the
protected area
SharedResource=8
Thread#12 is requesting Resource.
Thread#4 is requesting Resource.
Thread#4 has entered the
protected area
Thread#5 is requesting Resource.
Thread#10 is requesting Resource.
Thread#9 is requesting Resource.
Thread#6 is requesting Resource.
Thread#7 is requesting Resource.
Thread#7 is requesting Resource.
Thread#3 is requesting Resource.
Thread#5 is requesting Resource.
Thread#5 is requesting Resource.
Thread#6 is requesting Resource.
Thread#6 is requesting Resource.
Thread#6 is requesting Resource.
Thread#5 is requesting Resource.
Thread#9 is requesting Resource.
Thread#10 is requesting Resource.
Thread#12 is requesting Resource.
Thread#11 is requesting Resource.
Thread#8 #1 Time requests Resource.
Thread#8 is requesting Resource.
Thread#10 is requesting Resource.
Thread#6 is requesting Resource.
Thread#3 is requesting Resource.
Thread#8 is requesting Resource.
Thread#9 is requesting Resource.
Thread#8 is requesting Resource.
Thread#9 is requesting Resource.
Thread#8 is requesting Resource.
Thread#9 is requesting Resource.
Thread#9 is requesting Resource.
Thread#5 is leaving the
protected area
SharedResource=10
Thread#7 is requesting Resource.
Thread#11 is requesting Resource.
Thread#10 is requesting Resource.
Thread#7 is requesting Resource.
Thread#7 is requesting Resource.
Thread#5 is requesting Resource.
Thread#8 #2 Time requests Resource.
Thread#8 is requesting Resource.
Thread#8 is requesting Resource.
Thread#7 is requesting Resource.
Thread#5 is leaving the
protected area
SharedResource=11
Thread#6 is requesting Resource.
Thread#11 is requesting Resource.
Thread#10 is requesting Resource.
Thread#7 is requesting Resource.
Thread#3 is requesting Resource.
Thread#4 #1 Time requests Resource.
Thread#4 is requesting Resource.
Thread#4 is requesting Resource.
Thread#4 is requesting Resource.
Thread#4 is requesting Resource.
Thread#4 is requesting Resource.
Thread#3 is requesting Resource.
Thread#3 is requesting Resource.
Thread#3 is requesting Resource.
Thread#3 is requesting Resource.
Thread#3 is requesting Resource.
Thread#3 is entering the
protected area
SharedResource=12
Thread#5 is requesting Resource.
Thread#5 is requesting Resource.
Thread#5 is requesting Resource.
Thread#5 is requesting Resource.
Thread#5 is requesting Resource.
Thread#5 is requesting Resource.
Thread#10 is requesting Resource.
Thread#8 is requesting Resource.
Thread#3 is requesting Resource.
Thread#9 is requesting Resource.
Thread#12 is requesting Resource.
Thread#10 is requesting Resource.
Thread#7 is leaving the
protected area
SharedResource=12
Thread#6 is leaving the
protected area
SharedResource=12
Thread#6 is leaving the
protected area
SharedResource=12
Thread#6 is entering the
protected area
SharedResource=12
Thread#5 is leaving the
protected area
SharedResource=12
Thread#4 #1 Time requests Resource.
Thread#4 is requesting Resource.
Thread#4 is leaving the
protected area
Thread#9 is requesting Resource.
Thread#12 is requesting Resource.
Thread#6 is entering the
protected area
Thread#4 is entering the
protected area
Thread#12 is requesting Resource.
Thread#8 is requesting Resource.
Thread#10 is requesting Resource.
Thread#7 is requesting Resource.
Thread#9 is requesting Resource.
Thread#9 is requesting Resource.
Thread#9 is requesting Resource.
Thread#5 is requesting Resource.
Thread#12 is requesting Resource.
Thread#9 is requesting Resource.
Thread#5 is requesting Resource.
Thread#12 is requesting Resource.
Thread#4 is requesting Resource.
Thread#4 is requesting Resource.
Thread#4 is requesting Resource.
Thread#11 is requesting Resource. Thread#10 is requesting Resource.
Thread#3 is requesting Resource. Thread#9 is requesting Resource.
Thread#3 has entered the protected area Thread#7 is requesting Resource.
Thread#12 is requesting Resource. Thread#10 is requesting Resource.
Thread#7 is requesting Resource. Thread#11 is requesting Resource.
Thread#9 is requesting Resource. Thread#3 is requesting Resource.
Thread#10 is requesting Resource. Thread#11 is requesting Resource.
Thread#6 #3 Time requests Resource. Thread#6 is requesting Resource.
Thread#6 is requesting Resource. Thread#4 is requesting Resource.
Thread#4 is requesting Resource. Thread#8 is requesting Resource.
Thread#8 is requesting Resource. Thread#5 is requesting Resource.
Thread#5 is requesting Resource. Thread#11 is requesting Resource.
Thread#3 is leaving the protected area Thread#6 is requesting Resource.
Thread#6 is requesting Resource. Thread#6 has entered the protected area
Thread#7 is requesting Resource. Thread#10 is requesting Resource.
Thread#11 is requesting Resource. Thread#9 is requesting Resource.
Thread#9 is requesting Resource. Thread#8 is requesting Resource.
Thread#12 is requesting Resource. Thread#7 is requesting Resource.
Thread#8 is requesting Resource. Thread#4 is requesting Resource.
Thread#4 is requesting Resource. Thread#3 is requesting Resource.
Thread#12 is requesting Resource. Thread#3 is requesting Resource.
Thread#3 is requesting Resource. Thread#7 is requesting Resource.
Thread#10 is requesting Resource. Thread#9 is requesting Resource.
Thread#11 is requesting Resource. Thread#6 is requesting Resource.
Thread#11 is requesting Resource. Thread#12 is requesting Resource.
Thread#3 is requesting Resource. Thread#7 is requesting Resource.
Thread#9 is requesting Resource. Thread#8 is requesting Resource.
Thread#10 is requesting Resource. Thread#6 is requesting Resource.
Thread#4 is requesting Resource. Thread#8 is requesting Resource.
Thread#4 is requesting Resource. Thread#5 is requesting Resource.
Thread#5 is requesting Resource. Thread#11 is requesting Resource.
Thread#3 #2 Time requests Resource. Thread#3 is requesting Resource.
Thread#12 has entered the protected area Thread#5 is requesting Resource.
Thread#10 is requesting Resource. Thread#7 is requesting Resource.
Thread#9 is requesting Resource. Thread#10 is requesting Resource.
Thread#7 is requesting Resource. Thread#9 is requesting Resource.
Thread#11 is requesting Resource. Thread#11 has entered the protected area
Thread#12 is requesting Resource. Thread#11 is requesting Resource.
Thread#6 #4 Time requests Resource. Thread#6 is requesting Resource.
Thread#6 is requesting Resource. Thread#7 is requesting Resource.
Thread#5 is requesting Resource. Thread#10 is requesting Resource.
Thread#11 is requesting Resource. Thread#6 is requesting Resource.
Thread#11 is requesting Resource. Thread#12 has entered the protected area
Thread#12 is requesting Resource. Thread#6 is requesting Resource.
Thread#6 is requesting Resource. Thread#7 is requesting Resource.
Thread#5 is requesting Resource. Thread#10 is requesting Resource.
Thread#11 is requesting Resource. Thread#11 is requesting Resource.
Thread#7 is requesting Resource. Thread#12 is requesting Resource.
Thread#8 is requesting Resource. Thread#3 is requesting Resource.
Thread#3 is requesting Resource. Thread#7 is requesting Resource.
Thread#9 is requesting Resource. Thread#8 is requesting Resource.
Thread#10 is requesting Resource. Thread#3 is requesting Resource.
Thread#3 is requesting Resource. Thread#7 is requesting Resource.
Thread#9 is requesting Resource. Thread#8 is requesting Resource.
Thread#10 is requesting Resource. Thread#3 is requesting Resource.
Thread#3 is requesting Resource. Thread#7 is requesting Resource.
Thread#9 is requesting Resource. Thread#8 is requesting Resource.
Thread#10 is requesting Resource. Thread#3 is requesting Resource.
Thread#3 is requesting Resource. Thread#7 is requesting Resource.
Thread#9 is requesting Resource. Thread#8 is requesting Resource.
Thread#10 is requesting Resource. Thread#3 is requesting Resource.
Thread#3 is requesting Resource. Thread#7 is requesting Resource.
Thread#9 is requesting Resource. Thread#8 is requesting Resource.
Thread#10 is requesting Resource. Thread#3 is requesting Resource.
Thread#3 is requesting Resource. Thread#7 is requesting Resource.
Thread#9 is requesting Resource. Thread#8 is requesting Resource.
Thread#10 is requesting Resource. Thread#3 is requesting Resource.
Thread#3 is requesting Resource. Thread#7 is requesting Resource.
Thread#9 is requesting Resource. Thread#8 is requesting Resource.
Thread#10 is requesting Resource. Thread#3 is requesting Resource.
Thread#3 is requesting Resource. Thread#7 is requesting Resource.
Thread#9 is requesting Resource. Thread#8 is requesting Resource.
Thread#10 is requesting Resource. Thread#3 is requesting Resource.
Thread#3 is requesting Resource. Thread#7 is requesting Resource.
Thread#9 is requesting Resource. Thread#8 is requesting Resource.
Thread#10 is requesting Resource. Thread#3 is requesting Resource.
Thread#5 is requesting Resource.  
Thread#10 is requesting Resource.  
Thread#6 is requesting Resource.  
Thread#7 is requesting Resource.  
Thread#4 is requesting Resource.  
Thread#8 is requesting Resource.  
Thread#9 is requesting Resource.  
Thread#7 is requesting Resource.  
Thread#11 #2 Time requests  
Resource.  
Thread#11 is requesting Resource.  
Thread#8 is requesting Resource.  
Thread#4 is requesting Resource.  
Thread#10 is requesting Resource.  
Thread#6 is requesting Resource.  
Thread#3 is requesting Resource.  
Thread#9 is requesting Resource.  
Thread#11 is requesting Resource.  
Thread#8 is requesting Resource.  
Thread#12 is requesting Resource.  
Thread#11 has entered the  
protected area  
SharedResource=17  
Thread#5 is requesting Resource.  
Thread#5 has entered the  
protected area  
Thread#6 is requesting Resource.  
Thread#9 is requesting Resource.  
Thread#10 is requesting Resource.  
Thread#8 is requesting Resource.  
Thread#7 is requesting Resource.  
Thread#4 is requesting Resource.  
Thread#3 is requesting Resource.  
Thread#9 is requesting Resource.  
Thread#12 is requesting Resource.  
Thread#11 is requesting Resource.  
Thread#7 is leaving the  
protected area  
SharedResource=18  
Thread#8 is requesting Resource.  
Thread#8 is requesting Resource.  
Thread#12 is requesting Resource.  
Thread#10 is requesting Resource.  
Thread#10 has entered the  
protected area  
SharedResource=19  
Thread#10 #2 Time requests  
Resource.  
Thread#10 is requesting Resource.  
Thread#3 is requesting Resource.  
Thread#11 is requesting Resource.  
Thread#12 is requesting Resource.  
Thread#4 is requesting Resource.  
Thread#5 is requesting Resource.  
Thread#7 is requesting Resource.  
Thread#7 has entered the  
protected area  
SharedResource=20  
Thread#10 #2 Time requests  
Resource.  
Thread#10 is requesting Resource.  
Thread#3 is requesting Resource.  
Thread#11 is requesting Resource.  
Thread#12 is requesting Resource.  
Thread#10 has entered the  
protected area  
SharedResource=20  
...
3. Test results for the Dining Philosophers problem.

This is the result of using the new mutual exclusion algorithm to solve the Dining Philosophers problem. The numbers between dining events denote a philosopher's ID who is attempting to eat but failed.

Table is set
0
Philosopher: 0 is eating
2
Philosopher: 2 is eating
1
Philosopher: 1 is eating
3
Philosopher: 3 is eating
1
Philosopher: 1 is done
3
Philosopher: 3 is done
4
Philosopher: 4 is eating
4
Philosopher: 4 is done
VITA

Jingsong Zhang

Candidate for the Degree of

Master of Science

Thesis: A MUTUAL EXCLUSION ALGORITHM BASED ON REQUEST AND FAIR ACCESS

Major Field: Computer Science

Biographical:

Personal Data: Born in TianJin, China, November 23, 1969, son of Shibiao Chang and Yicun Chang

Education: Graduated from Nankai High school, TianJin, China, in June, 1988; received Bachelor of Science in Computer Science from Nankai University, TianJin, China, in July 1992; completed the requirements for the degree of Master of Science in Computer Science at the Computer Science Department of Oklahoma State University in December 2006.

Experience: Employed by Dell Inc., Round Rock, TX, as a Software Engineer Advisor, from June 2002 to August 2004. Employed by VERIZON Communication Corporation, Irving, TX, as a Sr. Consultant from August 2004 to June 2005. Employed by Perot System Corporation, Plano, TX, as a Software Development Specialist from June 2005 to present.
Scope and Method of Study: Process Synchronization is the set of techniques that are used to coordinate execution amongst processes. A common resource such as shared memory or a device may require exclusive access. In a multitasked/multithreaded system, processes have to coordinate amongst themselves to ensure that access is exclusive and fair. The mutual exclusion problem is one of the problems in inter-process communication that has been studied extensively. A number of mutual exclusion algorithms have been proposed. They have had varying degrees of success in handling the problem. These algorithms can be classified into hardware solutions and software solutions. The most well-known software solutions are turn-taking algorithms. This thesis gives an alternative mutual exclusion algorithm that is based on request and fair access instead of turn taking.

Findings and Conclusions: The algorithmic details and the implementation of the new mutual exclusion algorithm are given in this thesis report. The new algorithm’s implementation was tested on different multi-tasking multi-processor systems. The test environments consisted of a Dual CPU Dell PowerEdge 1800 running Windows 2003 Advanced Server and a four CPU HP ProLiant DL580 running Windows 2000 Server. The new algorithm was implemented in C#. The new algorithm was tested using classical dining philosophers and race condition problems. The test results showed that the new algorithm successfully solved these traditional mutual exclusion problems.