INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

UMI

A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor MI 48106-1346 USA
313/761-4700 800/521-0600

Distributed Deadlock-free Resource Allocation

by

Tanarug Issadisai

A dissertation submitted to the Graduate Faculty in Computer Science in partial fulfillment of the requirements for the degree of Doctor of Philosophy, The City University of New York

1998

This manuscript has been read and accepted for the Graduate Faculty in Computer Science in satisfaction of the dissertation requirement for the degree of Doctor of Philosophy.

Date	Chair of Examining Committee
Date	Executive Officer

Prof. Stathis Zachos

Dr. Jane Hsu

Supervisory Committee

THE CITY UNIVERSITY OF NEW YORK

TABLE OF CONTENTS

Part Three Algorithms

Part One

Introduction

Chapter 1

Background

The problem of deadlock detection and/or prevention in a distributed setting is an important one, and has received considerable attention. Dealing with deadlocks in a distributed environment is much harder than solving the problem in a (local) operating system. The main difficulty is the lack of centralized information showing the current state of the system.

The absence of a central repository means that, in general, the process requesting a resource must itself detect the possibility of deadlock, and if necessary take steps to prevent it.

There are two basic approaches to the problem:

1. Assign a centralized process to keep track of and control all resource allocation. In this method, all resource allocation activities have to be reported to the central process. The central process is the most important center of information. If the central process has a problem, all the information will be lost and the whole system will be in danger. In addition, this approach has the effect of 'serializing' the behavior of the system by reducing the concurrency. Although there may be system layouts and characteristics for which this would be an acceptable solution, in general this method suffers from lack of fault tolerance, poor availability, none ability to scale and bottleneck.

2. In a decentralized solution, which is more in the spirit of loosely coupled, semi-autonomous processes covering a large area, a process which wants to know about the deadlock situation will issue polling messages to all the processes and

analyze the deadlock situation from the answer to these messages. This method can be very costly because a process waiting for a resource might have to constantly poll the system to inquire whether there is the deadlock with the waiting resource. The question is when and how often the process should send out the polling message.

The deadlock is undesirable as the deadlock happens the system utilization is reduced: each process involved in the deadlock will be blocked, waiting for the resource to be available. The goal of this research is to find the way to reduce the damage from the deadlock by making the deadlock persistence time to be zero, and minimize the network communication.

To make the deadlock persistence time zero, we combine the resource allocation step with the deadlock detection step. By combining these two steps, we can solve the inefficiency problem of polling method.

Chapter 2

Introduction

There is a substantial amount of research in detecting distributed deadlock. Each publication on this topic is based on varying assumptions. Furthermore, there are many solutions for each assumption. But every one agrees that the basic problems are the absence of a global clock, the lack of centralized information, and detecting the distributed termination of the algorithms involved.

The global clock problem can be dealt with by implementing a virtual clock, but this in itself is an expensive, inefficient process.

There are two major ways of detecting the termination of the algorithm, one using reference counting and the other using markers or coloring.

Detecting distributed deadlock relies heavily on the algorithm for termination because the operation has to terminate. The repeated snapshot algorithm [1] is a basic algorithm that can be modified to guarantee that the operation will terminate.

Additionally, most of the published research is concentrated on the theoretical aspects and ignores the practical aspects important to real distributed systems. For example, the research about distributed deadlock does not mention how to collect the information from each process, how to recover from imminent deadlock, or the characteristics of the network connection between processes.

2.1 Well-known Distributed Deadlock Detection

There are several algorithms for distributed deadlock detection. Classified by algorithm, distributed deadlock detection can be grouped into four classes: path-pushing, edge-chasing, diffusion computation and global state detection[26].

2.1.1 A Path-Pushing Algorithm

In path-pushing deadlock detection algorithm, the detecting process keep collecting information from the network and update to local wait-for-graph (WFG). This algorithm has a problem because it can detect a phantom deadlock.

2.1.2 An Edge-Chasing Algorithm

In edge-chasing deadlock detection algorithm, the detecting deadlock process send the special message called a probe. A probe message will be send along the edges of the WFG and a deadlock is detected when a probe message returns to its

initiating process or repeat the same path.

2.1.3 A Diffusion Computational Algorithm

In a diffusion computational deadlock detection algorithm, deadlock detection computation is diffused through the WFG of the system. There are two types of messages used in diffusion computation, a query and a reply messages. A process can be in two states: active and blocked. A blocked process initiates deadlock detection by sending query messages to all the processes from whom it is waiting to receive a message (the dependent set).

When an active process receives a query or reply message, it ignores the message.

When a blocked process receives a query message, it performs the following actions: If the message is the first query message received from that process, then it propagates

the query to all the processes in its dependent set, else it returns a reply message to the sending query message process.

An initiator detects a deadlock when it receives reply messages from all the query messages it had sent out.

2.1.4 A Global State Detection Algorithm

In global state deadlock detection, the detecting deadlock process performs global state collecting information. There are several algorithms for collecting information that can be classified as single or multiple phases. The deadlock results from detecting WFG from the collected information.

Repeated snapshot is considered as a global state deadlock detection algorithm.

Chapter 3

Overview

This thesis paper provides a series of algorithms for resource allocation in a distributed environment. These algorithms use a combination of distributed deadlock detection and deadlock prevention. By combining deadlock detection and deadlock prevention, the system can achieve being a deadlock free system.

The resource allocation algorithms I am proposing do not use a central process and there is no need to poll the system for checking the deadlock. Additionally, deadlock prevention will be performed when a resource becomes available and will be assigned to a new process.

By using a method of sending and receiving markers as a tool, the major problem is ensuring that the algorithm will

terminate. Moreover, allowing multiple instances of the deadlock detection algorithm (executing on behalf of unrelated processes) overlapping in time is quite difficult. Additional, my proposing method can be enhanced to break the symmetric of resource requesting by impose the priority to the application process. In the event that deadlock or cyclic graph detected and there is more than one process involving in the same cyclic graph, the requesting of the process with highest priority will be accepted.

As the technology is heading to a cyber space technology, my resource allocation can be used as a tool for requesting the resource from the cyber space that can consider as a gigantic network without central information. There are a lot of applications that will benefit from this resource allocation are:

1. Video conferencing.
2. Cyber Space telephone/microphone.

3. Cyber Space speakers.

4. Cyber space printer/fax.

Part Two

Models

Chapter 4

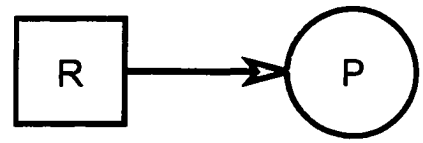
The Model

4.1 System Model

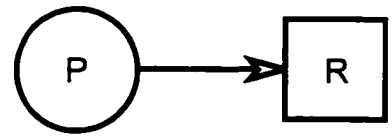
The system model consists of a finite set of application processes and resource processes. An application process may request to access a resource at any time. Upon the approval of the resource process, the application process has exclusive use of the resource. The application process will wait for the resource until the resource becomes available and eventually will release the resource.

We can capture the global state of the system (which is not available locally to any process) in terms of a bipartite, directed graph:

An edge directed from a resource/R to a process/P indicates that R is currently assigned to P.



An edge directed from a process/P to a resource/R signifies that P is currently waiting for R. P is then blocked.



To summarize, our system model has 3 out of the 4 conditions necessary for deadlock:

- **mutual exclusion**
- **no preemption**
- **hold-and-wait.**

In order to prevent deadlock we have to interfere with the fourth condition:

- **circular waiting.**

4.2 Network Model

Each process in the system is connected by an asynchronous message-passing network. Messages that are sent from the sender to the receiver are received reliably. A process has no knowledge about the network topology.

4.3 Resource Allocation Model

Application processes are service requester processes, resource processes are service provider processes. An application process may request the service from any resource processes. A resource process exclusively provides its service to only a single application process at a time. An application process will wait for the approval from the resource process unless it detects the deadlock condition. There are two characteristics for an application process to request the service from resource process.

- Execution is suspended during waiting for a resource (hold-and-wait)

- A resource can provide the service to only one application process (mutual exclusion)

We will look at two models:

1. Single Resource Allocation.

2. Multiple Resource Allocation.

4.3.1 Single Resource Allocation.

The application process can issue a request to only one resource at a time. If it needs more than one service, the application process has to request a service one by one in sequence and receive approval for one before requesting the next.

4.3.2 Multiple Resource Allocation.

The application process can issue a request to multiple resources at the same time. The application process will wait for the approval of all resources before returning control to the application task. If there is one request that causes the deadlock condition, the whole request will be canceled.

In order to achieve deadlock-free, we must combine three concepts together:

1. Deadlock detection
2. Deadlock resolution
3. Deadlock prevention

Chapter 5

The Problems

There are several problems to consider in the following areas.

5.1 Resource Allocation.

This is the basic problem dealing with in this thesis paper. We have considered the following aspects of our solution:

- Fairness of the algorithm.

  This problem had been taken cared by using first-come-first-serve algorithm. Any request that resource process received first will be granted if that grant do not create the deadlock.

  - Starvation consideration.

This problem is characterized by behavior of the running application. If any request had been rejected due to causing a deadlock, that request can not be started in near future. Additional, any acquired resource will be released eventually.

5.2 Deadlock Detection.

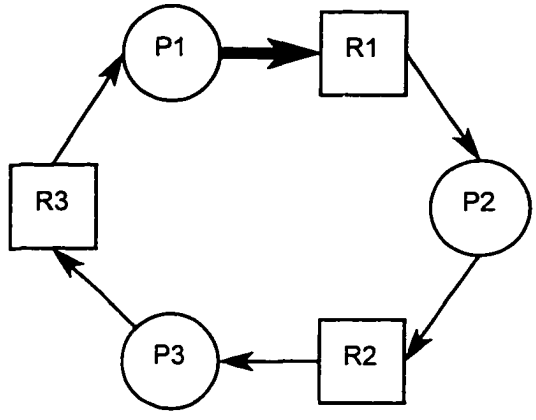
In a system which features

- Mutual exclusion,
- Hold-and-wait, and
- No preemption,

The only way to achieve deadlock-free operation is to avoid the possibility of circular waiting. In the absence of global knowledge, an application process that makes a request for a particular resource has no direct way to detect whether this request will close the loop, and introduce circularity. It is therefore inevitable that we must run an algorithm for the

requesting process to detect such a situation.

5.3 Deadlock resolution



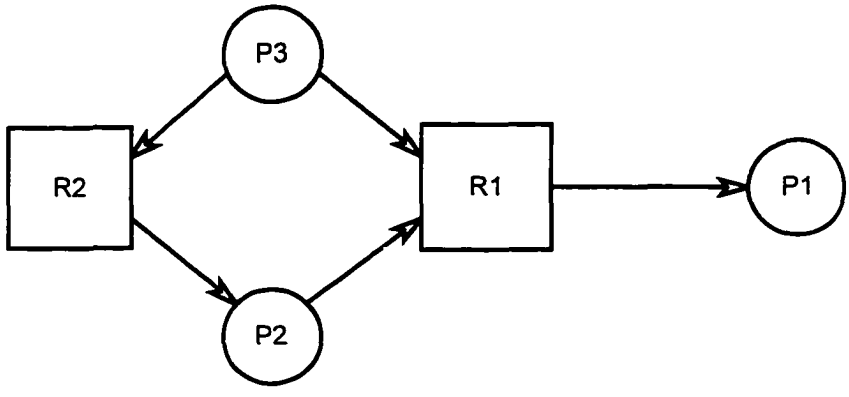
P1 requesting R1

When an application process detects that its request for a resource will possibly cause circular waiting, and thus create deadlock, it must take steps to avoid this situation. These steps involve, at the very least, canceling the request. However, this is not enough. The 'obvious' solution of trying the same request later will not work: all the other nodes i.e., processes and resources involved in this circularity are

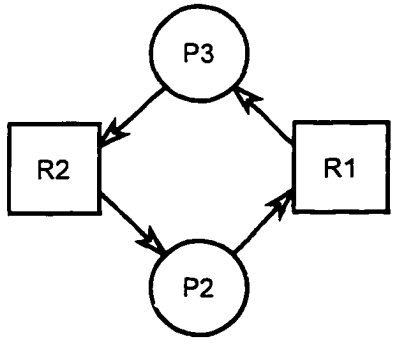
blocked, so that the same cycle will obtain 'at a later time' when the process renews its request.

The solution here is to force the process that detects a deadlock to back off and give up all of its acquired resources.

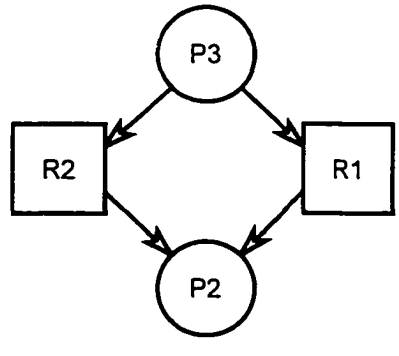
5.4 Deadlock Prevention.



Graph before resource R1 become free



R1 is assigned to P3



R1 is assigned to P2

When an application process requests an allocated resource, the resource process places the requesting application on its waiting list. If this particular request would cause a deadlock, the requesting process will detect this and withdraw the request. Otherwise, the resource will eventually become available to be allocated to one of the processes on its waiting list. At this point it is possible that this allocation (which will change the graph) will introduce circularity. Our algorithms have the property that there is at least one process in the waiting list that can be allocated the resource without introducing a cycle to the graph.

The problem then is to decide which of the waiting processes can be safely assigned the resource.

Part Three

Algorithms

Chapter 6

The Algorithms

Our algorithms exploit the following features of the allocation graph.

- In the stable state, the graph is an acyclic bipartite directed graph.

- The sink node (the root) can only be an application process.

- If there is a cyclic path, that path will be broken in the near future. (There are no stable cycles in the graph.)

- A node whose out-degree is greater than zero corresponds to a process which is blocked.

- The tree will expand by the sink node requesting another resource.

- The tree will shrink by the sink node releasing an acquired resource.

- If there is an incoming edge to a resource process, there has to be an outgoing edge (a state where the resource is free when there is an application process waiting for it is not stable).

- Resource nodes have out-degree no greater than one.

In our research, we consider two algorithms, depending on the constraints imposed on the way resources are requested.

- The 'simple' model forces applications to make requests one at a time.

- The 'general' model allows requests that can involve more than one resource. In this case, the requesting process blocks until all the resources that requested are assigned to it. If any of the resources involved would cause a deadlock, the application must cancel the request, as well as giving up all the resources assigned to it.

Chapter 7

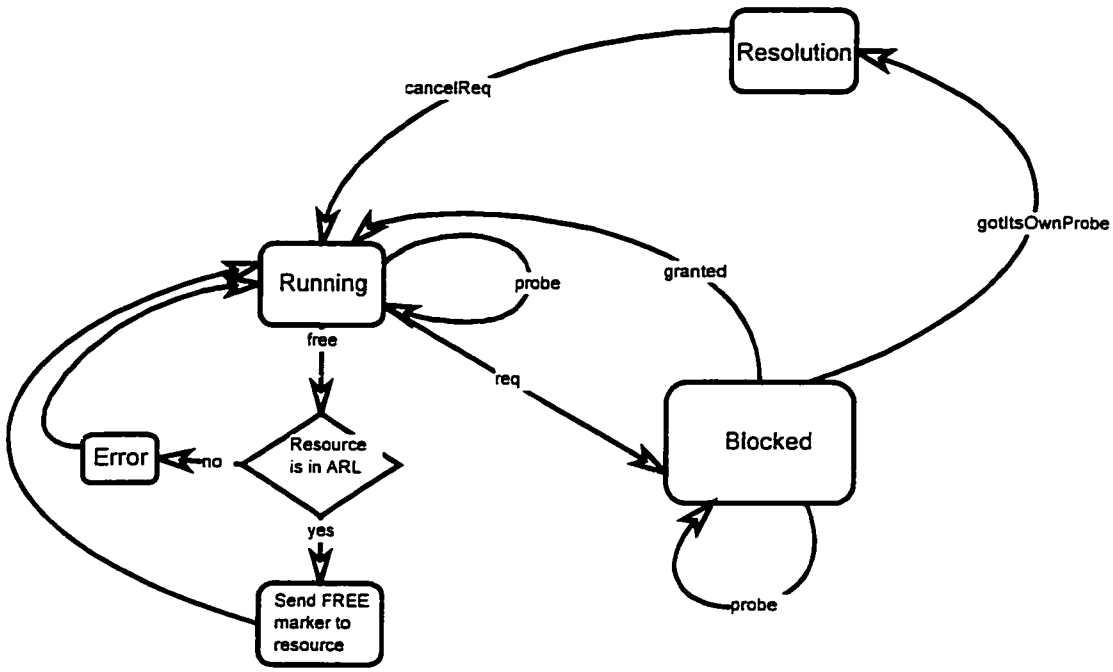
The Simple Model

In this model the allocation graph is a tree. A deadlock occurs when a process requests one of its parents. The algorithm is quite simple. In particular,

- The deadlock prevention step (when a resource that becomes free selects a process from its waiting list) is trivial. Any waiting process can be assigned the resource without fear of deadlock.

Consequently, this selection can be 'fair' so that the longest waiting process gets the resource. It is also possible to have this selection be driven by other considerations, such as priority.

7.1 Application Process.



State Diagram for Application Process

There are three states for application process:

1. *Running-state.* **This state, the application process has the control of the program execution.**

2. *Blocked-state.* **This state, the application process is in a waiting state. The application process will regain the control after all the requests had been granted.**

3. *Resolution-state.* **This state, the application process had detected a deadlock and start freeing all the allocated resource and canceling the requesting resource.**

***Running-State*.**

```
|| running; ifRequestOper -> ifRequestOper := false; running := false;
     blocked := true; oper := id U currentTime;
     res := removeFrist(reqList); R_res! reqResource(oper); reqTo := res;
```

When an application process want a service from a resource process, it send a request message to the resource process, set reqTo to the resource process and change to blocked-state.

```
|| running; ifFreeOper -> ifFreeOper := false;
   || isInList(arl, res) -> delete(arl, res); R_res!freeResource()
   || ~isInList(arl, res) -> errorMessage;
```

When an application process want to release a resource process, it send a release message to the resource process and remove resource process from ARL(Acquired Resource List).

|| running; $R_{i\ in\ act}$!probe(oper)-> SKIP;

When a running application process received a probe message, the message is ignored.

Blocked-state

```
|| R_reqTo?reqGrant() -> add(^arl, reqTo);
    [  || reqList = null -> running := true; reqTo := null;
       || reqList <> null -> oper := id U currentTime;
             res := removeFrist(reqList);
             R_res! reqResource(oper); reqTo := res;
    ];
```

When a blocked process received a grant message, it adds that resource to ARL. If there is more resource to request in `reqList` **then it send another request to the first resource in** `reqList` **otherwise it changes state to running-state.**

```
||_{i in arl} R_i?probe(oper) ->
  [  || ~isMyOper(oper) -> R_reqTo?probe(oper);
     || isMyOper(oper) -> deadLock := true; blocked := false;
          resolution := true;
  ];
```

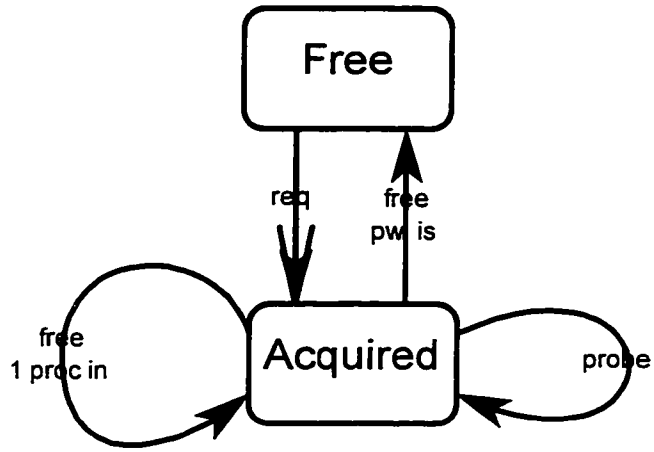
When a process received a probe message, it forwards to all the process that it is requesting. When a process received its own probe, it declares a deadlock and abort.

Resolution-state

```
|| resolution -> resolution := false; running := true;
    [  || R_reqTo!cancel() -> reqTo := null; reqList := null;
       ||_i in arl R_i!freeResource() -> delete(^arl, i);
    ];
```

When a process detected a deadlock, it frees all the acquired resources and cancels the requesting resource.

7.2 Resource Process



State Diagram for Resource Process

There are two states for a resource process:

1. *Free-state.* **This state, the resource process is free and ready to receive a request from an application process.**

2. *Acquired-state.* **This state, the resource process is assigned to an application process until it receives a free message.**

Free-state

|| free; $P_{i\,in}$.?reqResource(oper) -> assignTo := i; P_i!reqGrant();
free := false; assigned := true;

When a free resource received a request, it replies with a grant message, and changes its state to acquired-state.

Acquired-state

```
||P_in.?reqResource(oper) -> P_assignTo!probe(oper); add(^pwl, i);
```

When an acquired resource received a request, it add the requesting process to PWL(Process Waiting List) and send the probe message to the process that it is assigned to.

```
|| P_assignTo?freeResource()->
[
   || count(pwl) = 0 -> assignTo := null; free := true;
                        assigned := false;
   || count(pwl) >= 1 -> assignTo := removeFirst(pwl);
                         P_assignTo!reqGrant()
];
```

When an acquired resource received a release message, if there is no process waiting then it changes to a free state. If the PWL is not empty, it assigns to the first process in the PWL.

|| $P_{i \text{ in pwl}}$?probe(oper) -> $P_{assignTo}$!probe(oper)

When an acquired resource process received a probe message, it forwards that message to the application process that it is assigned to.

|| $P_{i \text{ in pwl}}$?cancel()-> delete(^pwl, i)

When an acquired resource process received a cancel message, it deletes that process from PWL.

Application Process.

```
P_app:: [  INIT; reqTo := null; arl := null;      /* arl = Acquire Resource
           running := true; blocked := false;

   || running; ifRequestOper -> ifRequestOper := false; running := false;
           blocked := true; oper := id U currentTime;
           res := removeFrist(reqList); R_res! reqResource(oper); reqTo := res;

   || running; ifFreeOper -> ifFreeOper := false;
      || isInList(arl, res) -> delete(arl, res); R_res!freeResource()
      || ~isInList(arl, res) -> errorMessage;

   || running; R_i in arl!probe(oper)-> SKIP;


   || blocked ->
      [  || R_reqTo?reqGrant() -> add(^arl, reqTo);
            [  || reqList = null -> running := true; reqTo := null;
               || reqList <> null -> oper := id U currentTime;
                     res := removeFrist(reqList);
                     R_res! reqResource(oper); reqTo := res;
            ];


         ||_i in arl R_i?probe(oper) ->
           [  || ~isMyOper(oper) -> R_reqTo?probe(oper);
              || isMyOper(oper) -> deadLock := true; blocked := false;
                   resolution := true;
           ];

      ];


   || resolution -> resolution := false; running := true;
         [  || R_reqTo!cancel() -> reqTo := null; reqList := null;
            ||_i in arl R_i!freeResource() -> delete(^arl, i);
         ];
```

Resource Process Algorithm.

```
R_ra::[
      pwl := null; assignTo := null;
      free := true; assigned := false;

   ...

   || free; P_i.in.?reqResource(oper) -> assignTo := i; P_i!reqGrant();
         free := false; assigned := true;

/* Should not happen
   || free; P_app?probe(oper) -> /* Should not happen */
*/

   || assigned ->
   [
      ||P_i.in.?reqResource(oper) -> P_assignTo!probe(oper); add(^pwl, i);
      || P_assignTo?freeResource()->
      [
         || count(pwl) = 0 -> assignTo := null; free := true;
                              assigned := false;
         || count(pwl) >= 1 -> assignTo := removeFirst(pwl);
                               P_assignTo!reqGrant()
      ];
      || P_i.in.pwl?cancel()-> delete(^pwl, i)
      || P_i.in.pwl?probe(oper) -> P_assignTo!probe(oper)
   ];
```

Chapter 8

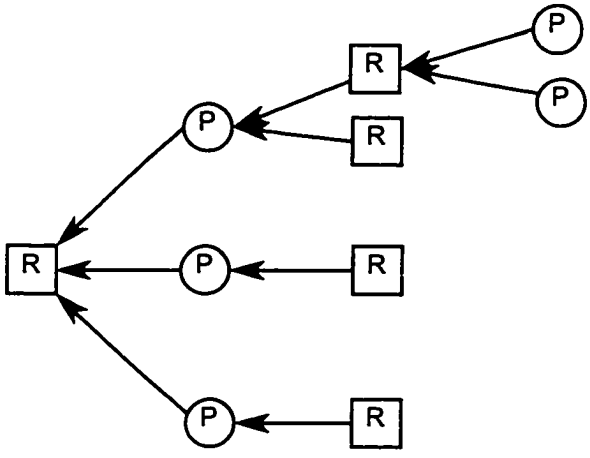
Proof of Simple Model

The proof of Simple Model had been divined into two parts:

1. **Deadlock Prevention disrequired.**

2. **Proof of Deadlock Detection for Simple Model.**

8.1 Deadlock Prevention Disrequired



Acyclic graph node N of out degree 0 can invert *any* incoming edge and maintain acyclic property (application process can have the degree of out bound edge only one).

Proof In order to two disjoint set of graph become a cyclic graph, two joining points are needed. Once node N invert one edge, it performs one joining point. Because every node can have only one out going edge, so it is impossible to have a joining point on the other side.

8.2 Deadlock detection

I have developed a complete set of proofs for the Simple algorithm. In particular, I have shown that, under the assumption that acquired resource will eventually be released, the Simple Algorithm has the following properties:

- An application process that issues a request is guaranteed to get a response.

- The response will be one of two things:
    - The request is granted, or
    - The request causes/involves a deadlock.

- A process P requesting resource R will never get a deadlock warning unless the allocation graph had a cycle involving P and R at some point during the interval between the time the request was issued and the time the warning was received.

Chapter 9

The General Model

In this model the allocation graph is a tree. A deadlock occurs when a process requests one of its parents. The algorithm is more complex. In particular,

- The deadlock prevention step (when a resource that becomes free selects a process from its waiting list) is required. Not any waiting process can be assigned the resource without fear of deadlock.

- There is no guarantee that the longest waiting process gets the resource. It is also possible to have this selection be driven by other considerations, such as priority.

Chapter 10

The General Model Basic Algorithm

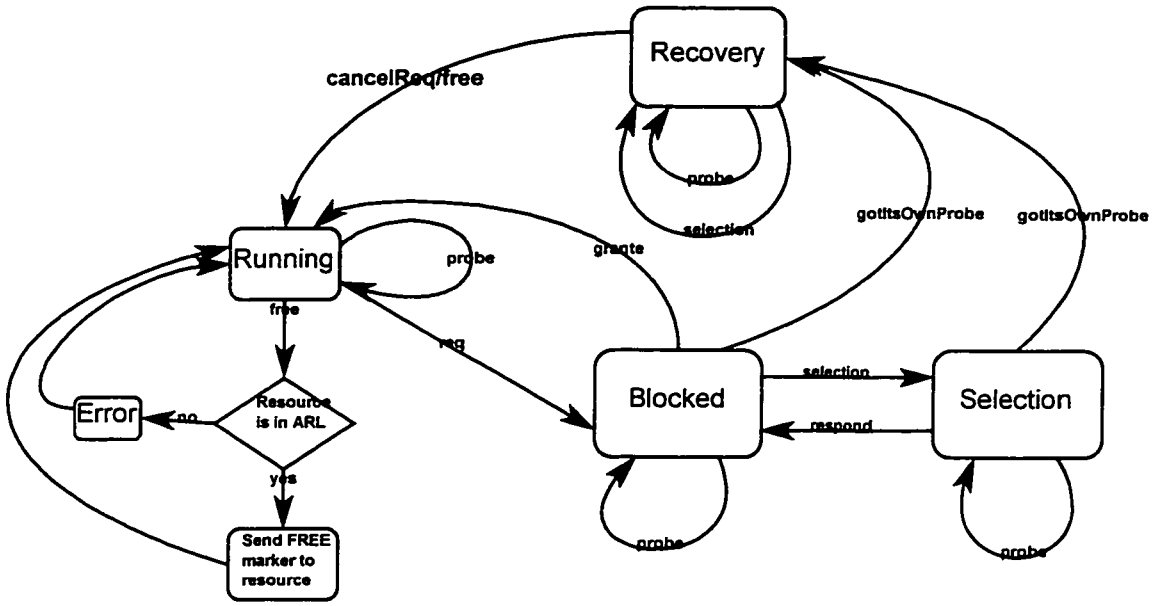
This model allows each application process issue more than one request simultaneously. The multiple requesting process will be suspended until all the requesting resources are granted.

There are two parts of the algorithm for the basic model:

1. Algorithm for the application process.

2. Algorithm for the resource process.

10.1 Application Process.



State Diagram for Application Process

There are four states for application process:

1. *Running-state*. **This state, the application process has the control of the program execution. When the system started, all the application process will be in this state.**

2. *Blocked-state*. **This state, the application process is in a waiting state. The control of the execution will return to application process after all the request had been granted.**

3. *Selection-state*. **This state, the application process is in a waiting state and there are some pending selection message that have not been responded. When all the selection messages had been responded, the application process will return to the blocked-state.**

4. *Recovery-state*. **When the application process received**

its own probe, it will move to this state. If there are any selections that are pending, it have to wait until all the selection had been complete, all the receiving selection message have to be responded. Then it cancels all the outstanding requests and frees all the acquired resource.

Remark. The selection-state starts from a blocked-state. There is no selection-state that started from a running state because of in the running-state when an application process received a selection message it will respond immediately.

Running-State.

```
|| running; ifRequestOper -> ifRequestOper := false; running := false;
     blocked := true; oper := id U currentTime;
     ||i in reqList Ri! reqResource(oper) -> add(^reqTo, i);
     reqList := null;
```

When an application process want a service from a resource process, it send a request message to the resource process and add the resource process to reqTo list and change to blocked-state.

```
|| running; ifFreeOper -> ifFreeOper := false;
   || isInList(arl, res) -> delete(arl, res); Rres!freeResource()
   || ~isInList(arl, res) -> errorMessage;
```

When an application process want to release a resource process, it send a release message to the resource process and remove resource process from ARL(Acquired Resource List) without changing the state.

|| running; $R_{i\ in\ arl}$!probe(oper)-> SKIP;

When a running application process received a probe message, the message is ignored.

|| running; $R_{i\ in\ arl}$!selection(oper)-> oper := oper U NORMAL; R_i!respond(oper);

When an application process received a selection message, it responds with the message with the normal return code.

Blocked-state

```
|| blocked; ~selection ->
      ||i in reqTo Ri?rSelection(oper) ->
           [  || count(reqTo) = 1 -> Ri!respond(oper U NORMAL);
              || count(reqTo) > 1 ->
                    sid := getSelectorIDFromOper(oper);
                    returnCode[sid] := NORMAL;      allSID = allSID U sid;
                    addOne(i.count[sid]); tmp := reqTo; delete(^tmp, i);
                    ||i in tmp Ri!selection(oper) -> subtractOne(i.count[sid]);
                    selection := true;
           ];
```

When an application process received a reverse-selection message (a message that resource process start sending out when that resource process start the selection algorithm). If there is no other requesting resource, it responds with normal return code, otherwise it forward the message to all other requesting resource process and move into selection state.

```
||i in reqTo Ri?reqGrant() -> moveChanFromTo(^reqTo, ^arl, i);
        [  || reqTo = null -> running := true;
           || reqTo <> null -> SKIP;
        ];
```

When a blocked process received a grant message, it move the granting resource process from reqTo to ARL. If it received all grants, it changes state to running-state.

```
||i in arl Ri?probe(oper) ->
  [  || ~isMyOper(oper) ->
        [  || ~isInList(seenOper, oper) -> add(^seenOper, oper);
                  ||i in reqTo Ri?probe(oper);
           || isInList(seenOper, oper) -> SKIP;
        ];
     || isMyOper(oper) -> deadLock := true;
        [  ||i in reqTo Ri!cancel() -> delete(^reqTo, i);
           ||i in arl Ri!freeResource() -> delete(^arl, i);
        ];
        blocked := false; running := true;
  ];
```

When a process received a probe message, it forward the probe message to all the process that it is requesting. When a process received its own probe, it declares a deadlock, cancel all the pending requests and free all the granted resource then abort.

```
||i in art Ri?selection(oper) -> selection := true;
  sid := getSelectorIDFromOper(oper); returnCode[sid] := NORMAL;
  allSID = allSID U sid; addOne(i.count[sid]);
  ||i in reqTo Ri!Selection(oper) ->subtractOne(i.count[sid]);
```

When a process received a selection messages. It forwards to all the requesting resource process and subtracts one to the counter. Note, as the application process is in a blocked-state that means the requesting resource process (reqTo) **is not empty.**

Selection-state

```
|| blocked; selection ->

    ||i in reqTo Ri?rSelection(oper) -> selection := true;
        sid := getSelectorIDFromOper(oper);
        addOne(i.count[sid]);
        [  || count(reqTo) = 1 -> oper := oper U NORMAL;
                    Pi!respond(oper); subtractOne(i.count[sid]);
           || count(reqTo) > 1  ->
                returnCode[sid] := NORMAL;
                allSID = allSID U sid; tmp := reqTo; delete(^tmp, i);
                ||i in tmp Ri!Selection(oper) -> subtractOne(i.count[sid]);
        ];
```

When an application process received a reverse-selection message (a message that resource process start sending out when that resource process start the selection algorithm), if there is no other requesting resource, it responds with normal return code, otherwise it forward the message to all other requesting resource process.

```
[  ||_{i in reqTo} R_i?reqGrant() -> /* R already send respond */
       moveChanFromTo(^reqTo, ^arl, i);
       [  || reqTo = null -> running := true; blocked := false;
                                 selection := false;
          || reqTo ~= null -> SKIP;
       ];
```

When a blocked process received a grant message, it moves the grating resource process from reqTo **to ARL. If it received all grants, it changes state to running-state.**

```
||_{i in arl} R_i?probe(oper) ->
  [  || ~isMyOper(oper) ->
     [  || ~isInList(seenOper, oper) -> add(^seenOper, oper);
              ||_{i in reqTo} R_i?probe(oper);
        || isInList(seenOper, oper) -> SKIP;
     ];
     || isMyOper(oper) -> deadLock := true;
     [  ||_{i in arl and j in allSID and i.count[j] > 0} ->
                  R_i!respond(returnCode[j]);
                        subtractOne(i.count[j]);
     ];
     [  ||_{i in reqTo} R_i!cancel()-> recovery := true;
        ||_{i in arl} R_i!freeResource() -> delete(^arl, i);
     ];
  ];
```

When a process received a probe message, it forward the probe message to all the process that it is requesting. When a process received its own probe, it declares a deadlock, cancel all the pending requests and free all the granted resource then

put itself into recovery mode.

```
||i in art Ri?selction(oper) ->
   sid := getSelectorIDFromOper(oper);
   addOne(i.count[sid]);
   [  || isSidInAllSID(allSID, sid) -> oper := oper U NORMAL;
                 Pi!respond(oper); subtractOne(i.count[sid]);
      || ~ isSidInAllSID(allSID, sid) -> returnCode[sid] := NORMAL;
           allSID = allSID U sid;
           ||i in reqTo Ri!Selection(oper) ->
                 subtractOne(i.count[sid]);
   ];
```

When a process received a selection message from an acquired resource process, it forward to all the requesting resource process if that selection message is new, otherwise it respond with the NORMAL code.

```
|| R_{i in reqTo}?respond(oper)-> sid := getSelectorIDFromOper(oper);
        subtract(i.count[sid]);
        [   || getReturnCode(oper) = LOOP -> returnCode[sid] = LOOP;
            || getReturnCode(oper) ~= LOOP -> skip;
        ];
        [       ^reqTo.count[sid] = 0 ->
                [   ||_{i in reqTo and i.count[sid] > 0} ->
                        R_i!respond(returnCode[sid]);subtractOne(i.count[sid]);
                        delete (^allSID, sid);
                ];
                i.count[sid] ~= 0 -> skip;
        ];
        allSID = null -> selection := false;
```

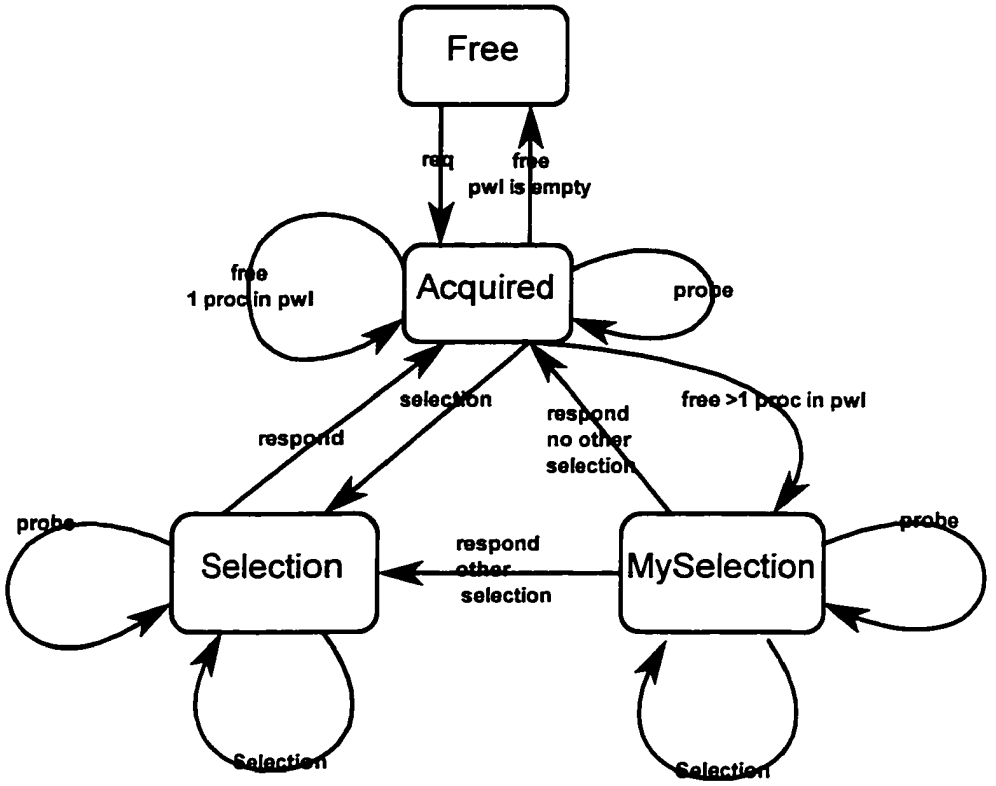
When a process received a respond message, if the return from respond message is 'LOOP', it sets the return code parameter for that selection to 'LOOP'. If the application process received all the respond from all the selection messages that send out, all the counter in reqTo list are zero, it send the respond back to the resource process that received the selection message from. If reqTo is empty then return to running-state otherwise return to blocked-state.

Recovery-state

```
|| recovery ->
   || R_{i in reqTo}?respond(oper)-> sid := getSelectorIDFromOper(oper);
            subtract(i.count[sid]); done := true;
            [  ||_{j in allSID and i.count[j] <> 0} -> done := false;
            ];
            done = true ->
            [
               delete(^allSID, sid); delete(^reqTo, i);
               allSID = null ->
               [
                  [  ||_{i in reqTo} R_i!cancel()-> delete(^reqTo, i);
                     ||_{i in arl} R_i!freeResource() -> delete(^arl, i);
                  ];
                  selection := false; recovery := false; blocked := false;
                  running := true;
            ];
];
```

When a recovery process receives a respond message, it checks whether all the selection messages had been respond. If all received selection messages had been responded, it cancel all the pending requests, free all the acquired resource and reset all the internal variables to initial value for running state.

10.2 Resource Process.



State Diagram for Resource Process

There are four states for resource process:

1. *Free-state*. **This state, the resource process is free and ready to receive a request from an application process.**

2. *Acquired-state*. **This state, the resource process is assigned to an application process until it receives a free message.**

3. *MySelection-state*. **This state, the resource process had received a free message and in a process of selecting one of application process in a process waiting list.**

4. *Selection-state*. **This state, the resource process had received a selection message from a waiting application process and forward the selection message to the application process that acquired that resource.**

Free-state

```
   || free; P_im.?reqResource(oper) -> assignTo := i; P_i!reqGrant();
        free := false; assigned := true;

/* Should not happen
   || free; P_app?probe(oper) -> /* Should not happen */
   || free; P_app?selection(oper) -> /* Should not happen */
*/
```

When a free resource received a request, it replies with a grant message, and changes its state to acquired-state.

Acquired-state

```
|| assigned; ~mySelection; ~selection ->
[
   ||P_i in .?reqResource(oper) -> P_assignTo!probe(oper); add(^pwl, i);
];
```

When an acquired resource process received a request, it add the requesting process to PWL(Process Waiting List) and send the probe message to the process that it is assigned to.

```
|| assigned; P_assignTo?freeResource()->
[
   || count(pwl) = 0 -> assignTo := null; free := true;
                        assigned := false;
   || count(pwl) = 1 -> assignTo := pwl; pwl := null;
                        P_assignTo!reqGrant()
   || count(pwl) > 1 -> assignTo := null; eligible := pwl;
        mySelection := true; oper := id U currentTime;
        ||_i in pwl P_i!rSelection(oper) -> subtractOne(i.count[id]);
];
```

When an acquired resource received a release message, if there is no process waiting then it changes its state to a free state. If there is only one process waiting, it assigns to that process otherwise it starts a my-selection by sending a reverse selection message to all the waiting application process.

|| assigned; $P_{i \text{ in pwl}}$?probe(oper) -> $P_{assignTo}$!probe(oper)

When an acquired resource process received a probe message, it forwards that message to the application process that it is assigned to.

```
|| assigned; P_{i in pwl}?selection(oper)-> selection := true;
        sid := getSelectorIDFromOper(oper);
        returnCode[sid] := NORMAL; allSID := sid;
        addOne(i.count[sid]); P_{assignTo}!Selection(oper);
        subtractOne(assignTo.count[sid]);
```

When an acquired resource process received a selection message, it changes to selection-state and forward that message to the application process that it is assigned to.

|| $P_{i \text{ in pwl}}$?cancel()-> delete(^pwl, i)

When an acquired resource process received a cancel message from the application process that is waiting, it deletes the process for process-waiting-list(pwl).

MySelection-state

|| $P_{i\,in}$.?reqResource(oper) -> add(^pReq, i); prOper[i] : = oper;

When a started-selection resource process received a request, it add the requesting process to a pending request List(pReq).

/* Never get freeResource()
|| $P_{assignTo}$?freeResource()-> */

A started-selection resource process will never received a free resource message.

```
|| P_{i in pwl}?cancel()->
    [  [  ||_{j in allSID and i.count[j] > 0} -> P_i!respond(returnCode[j]);
                                                 subtractOne(i.count[j]);
       ];
       delete(^pwl, i); delete(^eligible, i);
       pwl = null ->
       [  mySelection := false;
          moveChannelFromTo(^pReq, ^pwl);
          [
          || count(pwl) = 0 -> assignTo := null; free := true;
                                                 assigned := false;
          || count(pwl) = 1 -> assignTo := pwl; pwl := null;
                                                 P_{assignTo}!reqGrant()
          || count(pwl) > 1 -> assignTo := null;
                    mySelection := true; oper := id U currentTime;
                    ||_{i in pwl} P_i!rSelection(oper) ->
                                    subtractOne(i.count[id]);
          ];
       ];
    ];
```

When a started-selection resource process received a cancel message from a process in pwl, if there are any selection-messages that had not been respond, then respond those messages with NORMAL return code and delete the canceling process from pwl and eligible-list (eligible). If the remaining pwl is empty then move any process in pending-request-list (pReq) to pwl. If pwl is empty then changes its state to free state. If there is only one process in pwl then assign itself to that process. Otherwise, start a new selection process by sending a reverse selection message to all the application process in pwl.

```
|| P_{i in pReq}?cancel()->
    [
      [  ||_{j in pSID and i.count[j] > 0} -> oper := psOper[j] U NORMAL;
                                             P_i!respond(oper);
                                             subtractOne(i.count[j]);
      ];
      delete(^pReq, i); delete(^pProbe, i);
    ];
```

When a started-selection resource process received a cancel message from a process in pReq, **if there are any selection messages that had not been respond for that channel, then respond those messages with NORMAL return code and delete the canceling process from** pReq.

```
|| P_{i in pwl U pReq}?probe(oper) -> add (^pProbe, i); pOper[i] = oper;
```

When a started-selection resource process received a probe message, it adds the message to pending-probe (pProbe).

```
|| P_{i in pwl U pReq}?selection(oper)->
        sid := getSelectorIDFromOper(oper);
        addOne(i.count[sid]);
        [   || sid = id -> /* My Selection, and there is a LOOP */
                oper := oper U LOOP;
                P_i!respond(oper); subtractOne(i.count[sid]);
            || sid > id -> P_i!respond(oper U NORMAL);
                /* Resend selection because of obsolete info. */
                || isInList(^pwl, i) and i.count[id] = 0 ->
                                oper := id U currentTime;
                                P_i!rSelection(oper);
            || sid < id -> add(^pSID, i); psOper[i] = oper;
        ];
```

When a started-selection resource process received a selection message from the application process in pwl or pReq, if its own selection, it responds with the LOOP. If the selection from the higher priority resource process, it responds with NORMAL otherwise add the selection to pending-selection.

```
/* mySelection Mode, R waits a respond from PWL. */
|| P_{i in pwl}?respond(oper)->
        sid := getSelectorIDFromOper(oper);
        addOne(i.count[sid]); rCode = getReturnCode(oper);
        [  || rCode = LOOP -> returnCode[sid] = LOOP;
           || rCode = NORMAL -> SKIP;
        ];
        [  || sid = id -> /* My Selection */
             [
               [  || rCode = LOOP  -> delete(^eligible, i);
                  || rCode ~= LOOP     -> SKIP;
               ];
               [  || pwl[*].count[id] = 0 ->      /* All count in PWL is zero. */
                    [   i := getFirst(eligible);
                        assignTo := i;              /* Can I = NULL? */
                        P_i!grant(); mySelection := false;
                        moveChannelFromTo(^pReq, ^pwl);
                        [ ||_{i in pSID} ->
                           [ ||  i = assignTo -> oper = psOper[i] U NORMAL;
                                    P_i!respond(oper U NORMAL);
                             || i ~= assignTo -> oper = psOper[i];
                                    P_{assignTo}!selection(oper);
                           ];
                        ];
                        pSID := null; psOper := null;
                        [ ||_{i in pProbe} ->
                           [ ||  i = assignTo -> SKIP;
                             || i ~= assignTo ->
                                  ||_{j = 0 and j < prOper[i].count} ->
                                                oper := prOper[i][j];
                                                P_{assignTo}!probe(oper);
                           ];
                      pProbe := null; prOper := null;
                      ];

                  || pwl[*].count[id] ~= 0 -> SKIP;
               ];

          || sid > id -> P_i!respond(oper U NORMAL);
                         subtractOne(i.count[sid]);

          || sid < id -> add(^pSID, i); psOper[i] = oper;
        ];
```

When a started-selection resource process received a respond message, it calculates the new return code. If the returning code is LOOP, the new return code is set to LOOP otherwise no change.

If the receiving respond is for that resource process and if the return code is LOOP then delete the receiving application process that the respond message receive from eligible list.

If the resource process received all the respond from the application process then it assigned itself to the first application process in the eligible list. After that, all pending probe messages and pending selection messages will be forwarded to the application process that it is assigned to.

Selection-state

```
|| P_{i in pwl U pReq}?selection(oper)->
        sid := getSelectorIDFromOper(oper);
        addOne(i.count[sid]);
        [  || sid = id -> /* My Selection, and there is a LOOP */
                oper := oper U LOOP;
                P_i!respond(oper); subtractOne(i.count[sid]);
           || sid > id -> P_i!respond(oper U NORMAL);
                /* Resend selection because of obsolete info. */
                || isInList(^pwl, i) and i.count[id] = 0 ->
                                oper := id U currentTime;
                                P_i!rSelection(oper);
           || sid < id -> add(^pSID, i); psOper[i] = oper;
        ];
```

When a selection-state resource process received a selection message from a process in pwl **or** pReq**. If the priority of the selection message is greater than the priority of resource process, it responds that selection message with NORMAL, also, it checks whether that channel had received a respond message for its selection, if it received the respond, it have to re-send the reverse selection again. If the priority of the selection message is less than the priority of resource process, it adds that selection message to pending-selection-list** (pSID)**. Otherwise, the selection is its own selection message then it respond with LOOP.**

```
|| P_{i in pwl U pReq}?probe(oper) -> add (^pProbe, i); pOper[i] = oper;
```

When a selection-state resource process receives a probe message, it adds the probe message to a pending-probe-list (pProbe).

```
|| P_{assignTo}?respond(oper)-> sid := getSelectorIDFromOper(oper);
        addOne(assignTo.count[sid]);
        [  || getReturnCode(oper) = LOOP -> returnCode[sid] = LOOP;
           || getReturnCode(oper) ~= LOOP -> skip;
        ];
        [  || assignTo.count[sid] = 0 ->
              [  ||_{i in pwl and i.count[sid] > 0} ->
                        P_i!respond(returnCode[sid]);
                        subtractOne(i.count[sid]);
                        delete (^allSID, sid);
              ];
           || assignTo.count[sid] ~= 0 -> skip;
        ];
        allSID = null -> selection := false;
];
```

When a selection-state resource process received a respond message from the application process that it is assigned to, it calculates a new return code. If it had received all the respond for all selection messages that had send out, it responds with the new return code. If there is no selection pending then it set the selection flag to false.

The probe message is propagated as following:

1. For application process, the receiving probe message will be sent out to all the resource process that had been requested but have not been granted.

2. For resource process, the receiving probe message will be sent out to only the application process that it had been assigned to.

The selection message is propagated as following:

1. For application process, the receiving selection message will be remember the receiving channel and add the channel counter by one. The receiving message will be send out to every requesting resource process that have not been granted, and subtract the channel counter by one. Upon receiving a respond message, the channel counter will be added by one. The selection message will be respond when all the sending out message had been responded or all channel counters of out-channel are zero.

2. For resource process, the receiving selection message will be remember the receiving channel and add the channel counter by one. The receiving message will be send out to only the application process that it had been assigned to, and subtract the channel counter by one. Upon receiving a respond message, the channel counter will be added by one. The selection message will be respond when all the sending out message had been responded or all channel counters of out-channel are zero.

Resource Process.

```
R_ra::[
      id := myOwnPriority;
      pwl := null; assignTo := null;
      eligible := null;
      free := true; assigned := false;
      selection := false; mySelection := false;
      allSID := null;              /* All Selector ID */
      pReq := null;                /* Pending Request. */
      pSID := null;                   /* Pending Selection.*/
      pProbe := null;              /* Pending Probe. */

   ...

   || free; P_i in .?reqResource(oper) -> assignTo := i; P_i!reqGrant();
         free := false; assigned := true;

/* Should not happen
   || free; P_app?probe(oper) -> /* Should not happen */
   || free; P_app?selection(oper) -> /* Should not happen */
*/

   || assigned; ~mySelection; ~selection ->
   [
      ||P_i in .?reqResource(oper) -> P_assignTo!probe(oper); add(^pwl, i);
      || P_assignTo?freeResource()->
      [
         || count(pwl) = 0 -> assignTo := null; free := true; assigned := false;
         || count(pwl) = 1 -> assignTo := pwl; pwl := null; P_assignTo!reqGrant()
         || count(pwl) > 1 ->assignTo := null; eligible := pwl;
               mySelection := true; oper := id U currentTime;
               ||_i in pwl P_i!rSelection(oper) -> subtractOne(i.count[id]);
      ];
      || P_i in pwl?cancel()-> delete(^pwl, i)
      || P_i in pwl?selection(oper)-> selection := true;
               sid := getSelectorIDFromOper(oper);
               returnCode[sid] := NORMAL; allSID := sid;
               addOne(i.count[sid]); P_assignTo!Selection(oper);
               subtractOne(assignTo.count[sid]);
      || P_i in pwl?probe(oper) -> P_assignTo!probe(oper)
   ];
```

```
   || assigned; ~mySelection; selection ->
   [
      || P_i.?reqResource(oper) -> add(^pwl, i);

/* P send all respond before free the resource, so, should not be in selection
      || P_assignTo?freeResource()->
      [
         || count(pwl) = 0 -> assignTo := null; free := true; assigned := false;
         || count(pwl) = 1 -> assignTo := pwl; pwl := null; P_assignTo!reqGrant()
         || count(pwl) > 1 -> assignTo := null;
               mySelection := true; oper := id U currentTime;
               ||_i in pwl P_i!rSelection(oper);
      ];
*/

      || P_i in pwl?cancel()->
            [
                  /* Sending respond to the channel that had not responded yet. */
               [  ||_i in allSID and i.count[j] > 0 -> P_i!respond(returnCode[j]);
                                                      subtractOne(i.count[j]);
               ];
               delete(^pwl, i)
            ];

      || P_i in pwl?selection(oper)-> selection := true;
            sid := getSelectorIDFromOper(oper);
            addOne(i.count[sid]);
            [  || isSidInAllSID(allSID, sid) -> skip;/* Already got selection */
            [  || isSidInAllSID(allSID, sid) -> oper := oper U NORMAL;
                        P_i!respond(oper); subtractOne(i.count[sid]);
               || ~ isSidInAllSID(allSID, sid) -> returnCode[sid] := NORMAL;
                     allSID = allSID U sid;
                     P_assignTo!Selection(oper);
                     subtractOne(assignTo.count[sid]);
            ];

      || P_i in pwl?probe(oper) -> P_assignTo!probe(oper)

      || P_assignTo?respond(oper)-> sid := getSelectorIDFromOper(oper);
            addOne(assignTo.count[sid]);
            [  || getReturnCode(oper) = LOOP -> returnCode[sid] = LOOP;
               || getReturnCode(oper) ~= LOOP -> skip;
            ];
            [  || assignTo.count[sid] = 0 ->
                  [  ||_i in pwl and i.count[sid] > 0 -> P_i!respond(returnCode[sid]);
```

```
                                                    subtractOne(i.count[sid]);
                            delete (^allSID, sid);
                        ];
                    || assignTo.count[sid] ~ = 0 -> skip;
                ];
                allSID = null -> selection := false;
];
```

```
   || assigned; mySelection ->
   [
      || P_{i in .}?reqResource(oper) -> add(^pReq, i); prOper[i] := oper;
/* Never get freeResource()
      || P_{assignTo}?freeResource()-> */

      || P_{i in pwl}?cancel()->
           [  [  ||_{j in allSID and i.count[j] > 0} -> P_i!respond(returnCode[j]);
                                                        subtractOne(i.count[j]);
              ];
              delete(^pwl, i); delete(^eligible, i);
              pwl = null ->
              [  mySelection := false;
                 moveChannelFromTo(^pReq, ^pwl);
                 [
                 || count(pwl) = 0 -> assignTo := null; free := true;
                                      assigned := false;
                 || count(pwl) = 1 -> assignTo := pwl; pwl := null;
                                      P_{assignTo}!reqGrant()
                 || count(pwl) > 1 -> assignTo := null;
                         mySelection := true; oper := id U currentTime;
                         ||_{i in pwl} P_i!rSelection(oper) -> subtractOne(i.count[id]);
                 ];
              ];
           ];

      || P_{i in pReq}?cancel()->
           [
              [  ||_{j in pSID and i.count[j] > 0} -> oper := psOper[j] U NORMAL;
                                                      P_i!respond(oper);
                                                      subtractOne(i.count[j]);
              ];
              delete(^pReq, i); delete(^pProbe, i);
           ];

      || P_{i in pwl U pReq}?selection(oper)->
              sid := getSelectorIDFromOper(oper);
              addOne(i.count[sid]);
              [  || sid = id -> /* My Selection, and there is a LOOP */
                       oper := oper U LOOP;
                       P_i!respond(oper); subtractOne(i.count[sid]);
                 || sid > id -> P_i!respond(oper U NORMAL);
                       /* Resend selection because of obsolete info. */
                       || isInList(^pwl, i) and i.count[id] = 0 ->
                                                      oper := id U currentTime;
                                                      P_i!rSelection(oper);
```

```
        || sid < id -> add(^pSID, i); psOper[i] = oper;
    ];

|| P_inpwlUpReq?probe(oper) -> add (^pProbe, i); pOper[i] = oper;
```

```
/* mySelection Mode, R waits a respond from PWL. */
|| P_{i in pwl}?respond(oper)->
        sid := getSelectorIDFromOper(oper);
        addOne(i.count[sid]); rCode = getReturnCode(oper);
        [  || rCode = LOOP -> returnCode[sid] = LOOP;
           || rCode = NORMAL -> SKIP;
        ];
        [  || sid = id -> /* My Selection */
             [
               [  || rCode = LOOP  -> delete(^eligible, i);
                  || rCode ~= LOOP     -> SKIP;
               ];
               [  || pwl[*].count[id] = 0 ->      /* All count in PWL is zero. */
                    [   i := getFirst(eligible);
                        assignTo := i;              /* Can I = NULL? */
                        P_i!grant(); mySelection := false;
                        moveChannelFromTo(^pReq, ^pwl);
                        [ ||_{i in pSID} ->
                           [ ||  i = assignTo -> oper = psOper[i] U NORMAL;
                                    P_i!respond(oper U NORMAL);
                             || i ~= assignTo -> oper = psOper[i];
                                    P_{assignTo}!selection(oper);
                           ];
                        ];
                        pSID := null; psOper := null;
                        [ ||_{i in pProbe} ->
                           [ ||  i = assignTo -> SKIP;
                             || i ~= assignTo ->
                                  ||_{i = 0 and j < prOper[i].count} -> oper := prOper[i][j];
                                                          P_{assignTo}!probe(oper);
                           ];
                      pProbe := null; prOper := null;
                      ];

                  || pwl[*].count[id] ~= 0 -> SKIP;
               ];

          || sid > id -> P_i!respond(oper U NORMAL);
                         subtractOne(i.count[sid]);

          || sid < id -> add(^pSID, i); psOper[i] = oper;
        ];


];
```

Application Process.

```
P_app:: [  INIT; reqTo := null; arl := null;          /* arl = Acquire Resource
           running := true; blocked := false;
           seenOper := null;

  || running; ifRequestOper -> ifRequestOper := false; running := false;
           blocked := true; oper := id U currentTime;
           ||_{i in reqList} R_i! reqResource(oper) -> add(^reqTo, i);
           reqList := null;

  || running; ifFreeOper -> ifFreeOper := false;
     || isInList(arl, res) -> delete(arl, res); R_res!freeResource()
     || ~isInList(arl, res) -> errorMessage;

  || running; R_{i in arl}!probe(oper)-> SKIP;

  || running; R_{i in arl}!selection(oper)-> oper := oper U NORMAL; R_i!respond(oper);
```

```
|| blocked; ~selection ->
  [  ||i in reqTo Ri?reqGrant() -> moveChanFromTo(^reqTo, ^arl, i);
       [  || reqTo = null -> running := true;
          || reqTo <> null -> SKIP;
       ];

     ||i in arl Ri?probe(oper) ->
       [  || ~isMyOper(oper) ->
            [  || ~isInList(seenOper, oper) -> add(^seenOper, oper);
                    ||i in reqTo Ri?probe(oper);
               || isInList(seenOper, oper) -> SKIP;
            ];
          || isMyOper(oper) -> deadLock := true;
            [  ||i in reqTo Ri!cancel() -> delete(^reqTo, i);
               ||i in arl Ri!freeResource() -> delete(^arl, i);
            ];
            blocked := false; running := true;
       ];

     ||i in arl Ri?selction(oper) -> selection := true;
       sid := getSelectorIDFromOper(oper); returnCode[sid] := NORMAL;
       allSID = allSID U sid; addOne(i.count[sid]);
       ||i in reqTo Ri!Selection(oper) ->subtractOne(i.count[sid]);

     ||i in reqTo Ri?rSelection(oper) ->
       [  || count(reqTo) = 1 -> Ri!respond(oper U NORMAL);
          || count(reqTo) > 1 ->
                sid := getSelectorIDFromOper(oper);
                returnCode[sid] := NORMAL;  allSID = allSID U sid;
                addOne(i.count[sid]); tmp := reqTo; delete(^tmp, i);
                ||i in tmp Ri!selection(oper) -> subtractOne(i.count[sid]);
                selection := true;
       ];
  ];
```

```
|| blocked; selection ->
  [  ||_{i in reqTo} R_i?reqGrant() -> /* R already send respond */
        moveChanFromTo(^reqTo, ^arl, i);
        [  || reqTo = null -> running := true; blocked := false;
                              selection := false;
           || reqTo ~= null -> SKIP;
        ];

     ||_{i in arl} R_i?probe(oper) ->
        [  || ~isMyOper(oper) ->
           [  || ~isInList(seenOper, oper) -> add(^seenOper, oper);
                    ||_{i in reqTo} R_i?probe(oper);
              || isInList(seenOper, oper) -> SKIP;
           ];
           || isMyOper(oper) -> deadLock := true;
           [  ||_{i in arl and j in allSID and i.count[j] > 0} ->
                    R_i!respond(returnCode[j]); subtractOne(i.count[j]);
           ];
           recovery := true;
        ];

     ||_{i in arl} R_i?selction(oper) ->
        sid := getSelectorIDFromOper(oper);
        addOne(i.count[sid]);
        [  || isSidInAllSID(allSID, sid) -> oper := oper U NORMAL;
                    P_i!respond(oper); subtractOne(i.count[sid]);
           || ~ isSidInAllSID(allSID, sid) -> returnCode[sid] := NORMAL;
              allSID = allSID U sid;
              ||_{i in reqTo} R_i!Selection(oper) ->
                    subtractOne(i.count[sid]);
        ];

     ||_{i in reqTo} R_i?rSelection(oper) -> selection := true;
        sid := getSelectorIDFromOper(oper);
        addOne(i.count[sid]);
        [  || count(reqTo) = 1 -> oper := oper U NORMAL;
                    P_i!respond(oper); subtractOne(i.count[sid]);
           || count(reqTo) > 1  ->
                 returnCode[sid] := NORMAL;
                 allSID = allSID U sid; tmp := reqTo; delete(^tmp, i);
                 ||_{i in tmp} R_i!Selection(oper) -> subtractOne(i.count[sid]);
        ];
```

```
    || R_{i in reqTo}?respond(oper)-> sid := getSelectorIDFromOper(oper);
            subtract(i.count[sid]);
            [  || getReturnCode(oper) = LOOP -> returnCode[sid] = LOOP;
               || getReturnCode(oper) ~= LOOP -> skip;
            ];
            [     ^reqTo.count[sid] = 0 ->
                  [  ||_{i in reqTo and i.count[sid] > 0} ->
                        R_i!respond(returnCode[sid]);subtractOne(i.count[sid]);
                        delete (^allSID, sid);
                  ];
                  i.count[sid] ~= 0 -> skip;
            ];
            allSID = null -> selection := false;




|| recovery ->
    || R_{i in reqTo}?respond(oper)-> sid := getSelectorIDFromOper(oper);
            subtract(i.count[sid]); done := true;
            [  ||_{j in allSID and i.count[j] <> 0} -> done := false;
            ];
            done = true ->
            [
               delete(^allSID, sid); delete(^reqTo, i);
               allSID = null ->
               [
                  [  ||_{i in reqTo} R_i!cancel()-> delete(^reqTo, i);
                     ||_{i in arl} R_i!freeResource() -> delete(^arl, i);
                  ];
                  selection := false; recovery := false; blocked := false;
                  running := true;
            ];
];
```

Chapter 11

Properties

Distributed Deadlock-Free Resource Allocation.

As the network system started. The state of the network is in a save state. The state of the network is changed, when the application process issues a request/free for a resource/resources, and when the resource is granted to an application process. On the other hand, the state of the network is changed when the edged of the graph is changed.

When the state of network is changed, there are two possibilities to happen. One is the network still maintain its save state, the other is in the state of the deadlock. Because of this reason, when the application process issue the request, the internal system has to perform deadlock detection algorithm by sending the probe message. If that request causes the

deadlock, the requesting application process has to be aborted and releases all the acquired resources. This requirement is needed because just only rejected the request does not make the network out of the potential deadlock situation. The application process could try to request the same resource and constantly is rejected.

As the deadlock state is a stable state, therefore the probe message (Deadlock Detection Message) that generated by a requester process will be traveled back to the original process which will detect the deadlock when the process receives its own probe message.

Additional, by the property of wait-for-graph, the graph can expand by two conditions. First the application process request the allocated resource in the graph. The second, the sink node issues the request to another resource process. In the worse case situation, there will be at least one sink node. If that sink node request a resource which is already in the tree, the

deadlock will occur but the system will break the deadlock.

In Deadlock Detection Algorithm, the algorithm does not need to assign a state to the process. When a process requests a resource, it assumes that its request does not create a deadlock and that request will be granted when the resource is available. If the deadlock is caused by that request and the deadlock state is a stable state, eventually the probe message will travel back to the original process and the deadlock condition will be detected and the request will be canceled which eliminate the deadlock.

Deadlock Prevention is needed when a resource become free and that resource try to assign itself to one of a process in process waiting list. Because of the assignment itself to a new process have to be done after the algorithm is complete, therefore the deadlock prevention have to have a signal to show that algorithm had been terminated. The signal of termination is defined by all the tokens that send out are replied. So, each edge in the graph will be recorded in two ways, add one to the channel counter when receive an incoming message and subtract one to the channel counter when send the outgoing message. The multiple deadlock prevention can be handled by adding TAG or another dimension to all the channel counter. In multiple deadlock prevention, the priority of the resource is introduced to break the symmetric condition when there more than one resource process on the same circle-wait-for-graph, are freed.

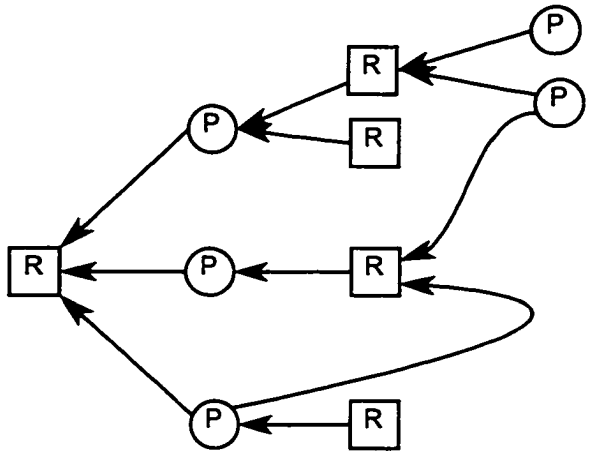
Base on the knowledge of sending the message, the state of each process is recorded as started and stopped. So, the deadlock

prevention can be repeated.

In the real situation, the deadlock detection and deadlock prevention can happen concurrently. These two algorithms are working independently. Because the deadlock prevention need to remember the state of each edge, so any change of the edge during the deadlock prevention will be deferred or remembered and be applied when the deadlock prevention terminated. As I had mentioned, the edge changing is caused by two situations. The first situation when the sink node make a request. The second situation when a resource requests a resource on the graph.

The total network will never be in a permanent state of deadlock, when the last sink-node make a request which could request a free resource or request an acquired resource. The first case the request will be granted and the last sink-node will continue the operation. In the second case, the new

request will be under the deadlock detection and the deadlock situation will be detected and the sink-node has to cancel the request and release all the acquired resources. So, in the total network, there will be at least one process continue its operation.



Theorem Acyclic graph node n of out degree 0 can invert one edge and maintain acyclic property.

Axiom1. If we invert an edge and then create a wait-for graph incoming from node P then if we invert the edge to node P there will be no wait-for graph.

Axiom2. If a process can wait for only one resource then there will be no cyclic graph. Then each path will be a disjoint partition.

Proof If we invert an edge and then there is a cyclic wait-for

graph. We can order the node in such a way that

$$\text{If} \quad Na \xrightarrow{path} Nb \quad \text{then}$$

$$Na < Nb$$

From Axiom1. $Nb \xrightarrow{no\ path} Na$ then

$$Nb \text{ not } < Na$$

Given condition, there is no cyclic wait-for graph before invert the edge.

From given condition, we can create a partial order list of node. The node that is at the end of the list will be the node that there is no path from that to the previous order node. So, the node at the end of the list can be invert.

When the system just started. All the resource processes are free. None of the application process had acquired any resource. Each process is a graph by itself.

State of each graph is changed by three reasons:

1. Application Process requests a resource/resources.
2. Application Process frees a resource.
3. Resource Process select to grant to a resource process.

Application process requests a resource/resources. This action will cause two or more graph to combine into one graph. The number of graph on the system will be reduced. If the requested resource process is assigned to another process, probe algorithm will have to be performed to detect whether there is the deadlock cause by this request.

Application process frees a resource. This action will cause a graph split into two graphs. The number of graph on the system

will be increased.

Resource process select to grant to an application process. This action causes no combining or splitting the graph. It causes an edge to change the direction. So, the number of graph on the system will remain the same. If there are more than one application process waiting, the selection algorithm has to be performed to eliminate the edge that could cause a deadlock after the edge had change the direction.

Chapter 12

Evaluation and Comparison of the Algorithms

The Simple Algorithm is conceptually cleaner and simpler than the General Algorithm is. General Algorithm allows an application process making a multiple-resource request that introduces a complicated case.

Obviously, the general algorithm seems to be faster than the simple algorithm in requesting the resource because it can request more than one resource at a time. But it introduces a higher probability of deadlock. At the same time, the general algorithm needs a completion of selection algorithm before it can make a decision in allocated the resource. This causes a decreasing the efficiency of the algorithm.

So, the efficiency of these two algorithms will depend on the load of the system, the number of resource processes

available for the entire system also the number of resources that an application process requests each time.

$$E = f(R,\ L,\ M)$$

E = Efficiency of the algorithm

R = Requesting rate.

$$R = \frac{Number\ of\ Application\ Process\ start\ a\ request}{Time}$$

L = Load factor

$$L = \frac{Number\ of\ Resource\ Process}{Number\ of\ Application\ Process}$$

M = Multiple request factor.

$$M = \frac{Number\ of\ requested\ resource}{Number\ of\ requesting}$$

M = 1 **for simple algorithm**

M >= 1 **for general algorithm**

In case, $M_g = 1$ for general algorithm, the efficiency will be the same as simple algorithm because M_s for simple algorithm always equal one.

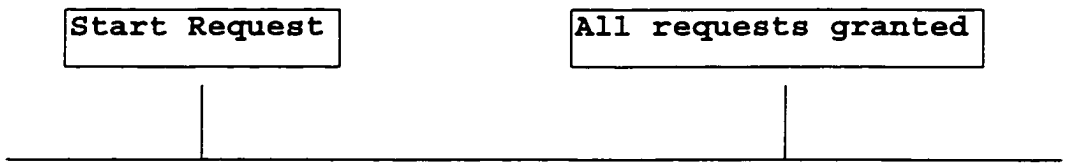
In case, $M_g > 1$ for general algorithm, the efficiency will be better than simple algorithm.

For simple algorithm, the bigger L_s will not improve the efficiency because the resource requesting is done in serialize. But in vice versa for general algorithm, because in general algorithm, resource requesting are done in parallel and the higher L_g, the lower the chance that resource has to do selection algorithm.

Chapter 13

Simulation

The objective of the simulation is to find the difference of efficiency of the two algorithms. In order to compare the efficiency of these two algorithms, I have to assume that all other factors are fixed. Differences only that, the simple algorithm, the application process can request more than one resource at a time.



As, the activities of the two algorithms different only between the application process started the request and received grant from all the requested resource. So, the framework of the simulator is to create a fixed pattern of all the application

process that requests the resource.

For the simple algorithm, the request for more than one resource, the application process will request one resource wait until received the grant and request the next resource, doing so until receive all grant then return the control to the application.

Assumption, the built-in random function is perfectly normal distribution.

The simulator that I had built base on following algorithms:

Assigned Variable:

MAX_USE_RESOURCE = Maximum tick in requiring a resource

NRES = Number Resource Processes

NPRO = Number Application processes

So, $L = \frac{NRES}{NPRO}$

R = Maximum tick that a process will idle before request a new resource.

```
Tick = current time
For I = 1 to NPROC
     startReq(I) = Tick + (random() mod R)
     numReq(I) = getNumRequest(numAcquiredResource)
end
```

So, array of startReq will contain the time that the application resource will start the request and array of numReq will contain the number of resource that it will request. Function, getNumRequest, will return the number of resources that the application process will request. The return number is based on the status of the application process and the random number generator. The status of application process will be how many resource processes that the application process had acquired. The higher number of acquired the resource, the lower the number of resource for the current request. With the number

of acquired resources and the random number, the function will use this information to lookup the internal table to get the number of current request.

The main Loop.

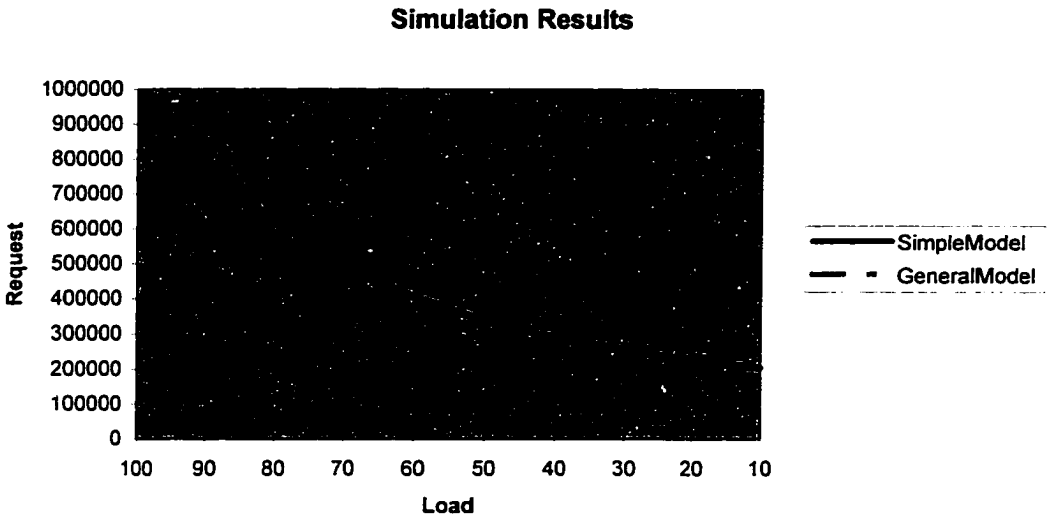
```
Do forever
     For I = 1 to NPROC
          If  TICK = startReq(I) then
               Request the resource for numReq(I) for j tick
          End if
     End for
     For I = 1 to NRES
          If TICK = freeResource(I) then
               Free the resource(I)
          End if
     End for
     TICK = TICK + 1
End do
```

When all requesting resource granted, freeResource(I) will be set to the time that the resource will be deallocated.

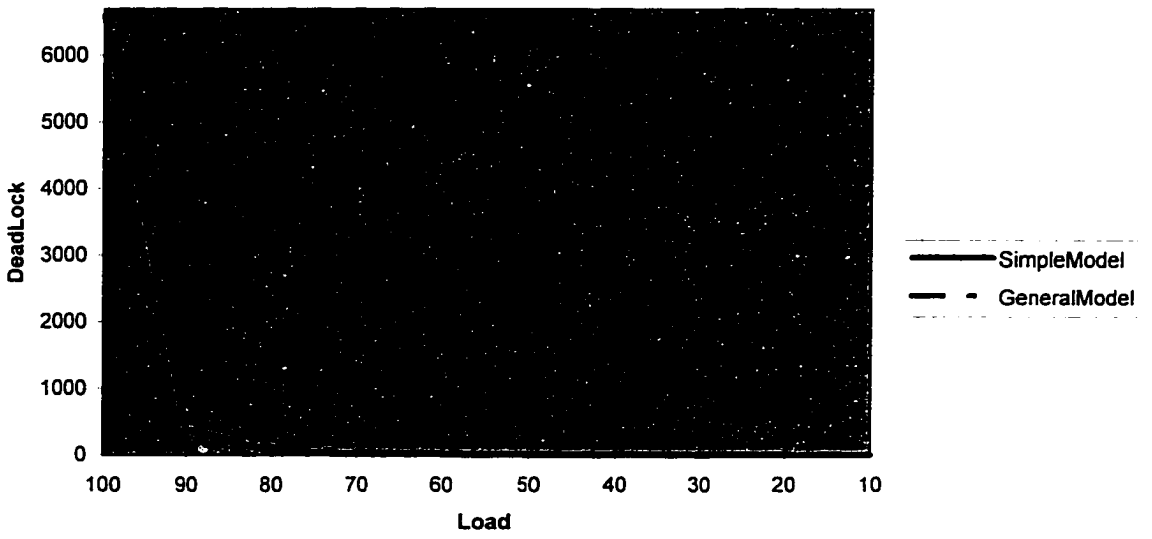
```
For I = 1 to number of requesting resource
     j = random() % MAX_USE_RESOURCE
     freeResource(I) = TICK + j
End for
```

Probability Table

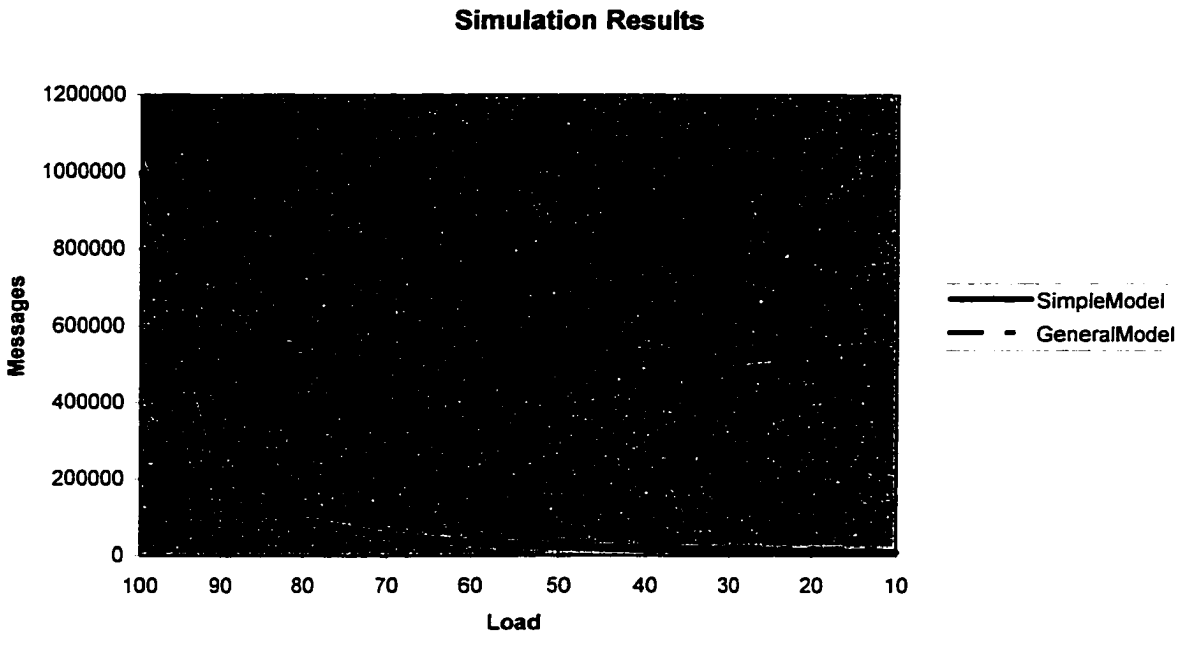
Acq\Req	1	2	3	4	5	6	7	8	9	10
0	0.2	0.2	0.2	0.15	0.1	0.05	0.04	0.03	0.02	0.01
1	0.3	0.2	0.15	0.1	0.1	0.05	0.04	0.03	0.03	0
2	0.3	0.2	0.15	0.1	0.1	0.08	0.04	0.03	0	0
3	0.31	0.25	0.15	0.1	0.1	0.05	0.04	0	0	0
4	0.35	0.25	0.15	0.1	0.1	0.05	0	0	0	0
5	0.4	0.25	0.15	0.1	0.1	0	0	0	0	0
6	0.5	0.25	0.15	0.1	0	0	0	0	0	0
7	0.6	0.25	0.15	0	0	0	0	0	0	0
8	0.85	0.15	0	0	0	0	0	0	0	0
9	1	0	0	0	0	0	0	0	0	0
10	1	0	0	0	0	0	0	0	0	0
11	1	0	0	0	0	0	0	0	0	0
12	1	0	0	0	0	0	0	0	0	0
13	1	0	0	0	0	0	0	0	0	0
14	1	0	0	0	0	0	0	0	0	0
15	1	0	0	0	0	0	0	0	0	0
16	1	0	0	0	0	0	0	0	0	0
17	1	0	0	0	0	0	0	0	0	0
18	1	0	0	0	0	0	0	0	0	0
19	1	0	0	0	0	0	0	0	0	0
20	1	0	0	0	0	0	0	0	0	0



Result from the simulation shown that the General Model give a same result as the Simple Model until the load had increase significantly then the General Model starts to perform worse than the Simple Model.



Result from the simulation shown that the Simple Model causes about the same deadlock as the General Model. When the load is low, the number of deadlock is almost flat. When the load is significantly increased, the number of deadlock is enormously increased.



Result from the simulation shown that the General Model generates more messages than the Simple Model when the load is high and about the same when the load is low.

Conclusion.

From the results, we can conclude that the General Model does not perform better than the Simple Model. The fact the Simple Model performs better than the General Model performs when the load is high.

Chapter 14

Summary

The objective of this thesis is to combine the deadlock detection with resource allocation to reduce the network persistence time. Additional, the deadlock detection had been designed to be able to operate in online environment as the algorithm can be repeated and there is no freezing the operation.

In the real situation, the depth of the allocation tree is not deep. The more the depth of the tree, the more deteriorate of system performance. But controlling the depth of allocation tree is very difficult because it depend on the application. What the system analyst could do is to monitor the utilization of resource. If the utilize is very high that means that resource could be a bottleneck.

Chapter 15

Open Problems

Base on this thesis. I had introduced a new algorithm that combined resource allocation and deadlock detection together. But the problems in the area of deadlock detection and resource allocation are vastly enormous. So, there are a number of the problems that I did not cover. The open problems are following:

1. If there is more than one pending request that involved with the same loop, all the pending requests will detect a deadlock and all of them will be aborted. This problem enhanced by assigning a unique priority to all application processes. Only the lowest priority will detect the deadlock and aborted.

2.As the selection algorithm need the completion of the process

before the reassignment can be done, which caused the resource idle for a long time. This problem enhanced by the resource process that started the selection assigned itself to the first respond that does not cause the loop and let the algorithm terminated later.

3. The capability of combining algorithm between simple algorithm and general algorithm. There is no need to use selection algorithm for the graph that the maximum fan out of application process is degree 1.

References.

1. Luc BOUGE', Repeated snapshots in distributed systems with synchronous communications and their implementation in CSP, Theoretical Computer Science 49(1987) 145-169

2. S.D. Brookes and A.W. Roscoe, Deadlock analysis in networks of communicating processes, Distributed Computing (1991) 4: 209-230

3. S.C Shyu, Victor O.K. Li and C.P. Wang, An Abortion-Free Distributed Deadlock Detection/Resolution Algorithm, IEEE Transactions on Software Engineering, (1990), 167-174

4. R.Obermarck, Distributed Deadlock Detection System, ACM Transactions on Database Systems, Vol7, No. 2, June 1982

5. Walter Kohler, A Survey of Techniques for Synchronization and Recovery in Decentralized Computer Systems, Computing

Surveys, Vol 13, No. 2, June 1981

6. Peter M. G. Apers, Data Allocation in Distributed Database Systems, ACM Transactions on Database Systems, Vol 13, No. 3, Sept 1988.

7. K. Mani Chandy and Leslie Lamport, Distributed Snapshots: Determining Global States of Distributed Systems, ACM Transactions on Computer Systems, Vol. 3, No. 1, February 1985.

8. Marta Rukoz, Hierarchical Deadlock Detection for Nesting Transactions, Distributed Computing (1991) 4:123-129

9. Mani Chandy and Jayadev Misra, of Distributed Programs: Quiescense Detection, ACM Transactions on Programming Languages and Systems, Vol. 8, No. 3, July 1986, 326-343

10. H. Conrad Cunningham and Gruia-Catalin Roman, A UNITY-Style Programming Logic for Shared Dataspace Programs, Transactions

on Parallel and Distributed Systems, Vol. 1, No. 3, July 1990

11. Jean-Michel Helary, Observing Global States of Asynchronous Distributed Applications, Distributed Algorithms, Lecture Notes in Computer Science 392, Springer-Verlag, New York

12. Jean - Marc Jezequel, Building a Global Time on Parallel machines, Distributed Algorithms, Lecture Notes in Computer Science 392, Springer-Verlag, New York

13. Beverly A. Sanders, Distributed Deadlock Detection and Resolution with Probes, Distributed Algorithms, Lecture Notes in Computer Science 392, Springer-Verlag, New York

14. Ouri Wolfson, The Virtues of Locking by Symbolic Names, Journal of Algorithms 8, 536 - 556 (1987)

15. Edsger W. Dijkstra, C.S. Scholten, Termination Detection for Diffusing Computations, Information Processing Letters, 1-4

(29 August 1980)

16. Edsger W. Dijkstra, W.H.J. Feijen and A.J.M. van Gasteren, Derivation of a Termination Algorithm for Distributed Computations, Information Processing Letters, 217-219(10 June 1983)

17. Mani Chandy and Jayadev Misra, Parallel Program Design a Foundation, Addison-Wesley Publishing Company, New York, 1988

18. Shing-Tsaan Huang, A Distributed Deadlock Detection Algorithm for CSP-Like Communication, ACM Transactions on Programming Languages and Systems, Vol. 12, No. 1, January 1990, 102-122

19. Barbara Liskov and Robert Scheifler, Guardians and Actions: Linguistic Support for Robust, Distributed Programs, ACM Transactions on Programming Languages and Systems, Vol. 5, No. 3, July 1983, 381-404

20. Richard N. Taylor, A General-Purpose Algorithm for Analyzing Concurrent Programs, Communications of the ACM, Vol. 26, May 1983, 226-240

21. Peter Wegner and Scott A. Smolka, Processes, Tasks, and Monitors: A Comparative Study of Concurrent Programming Primitives, IEEE Transactions on Software Engineering, July 1983, 446-462

22. Kristine Stougaard Thomsen and Jorgen Lindskov Knudsen, A Taxonomy for Programming Languages with Multisequential Processes, Journal of Systems and Software, Vol. 7, 1987, 127-140

23. Mukesh Singhal, Deadlock Detection in Distributed Systems, COMPUTER, November 1989, 37-47

24. Gary S. Ho and C. V. Ramamoothy, Protocols for Deadlock

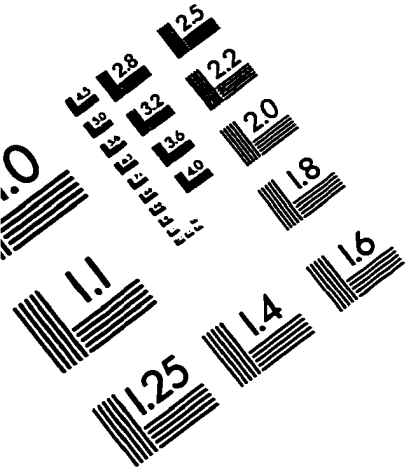
Detection in Diistributed Database Systems, IEEE Transactions on Software Engineering, Vol. SE-8, NO. 6, November 1982, 554-557

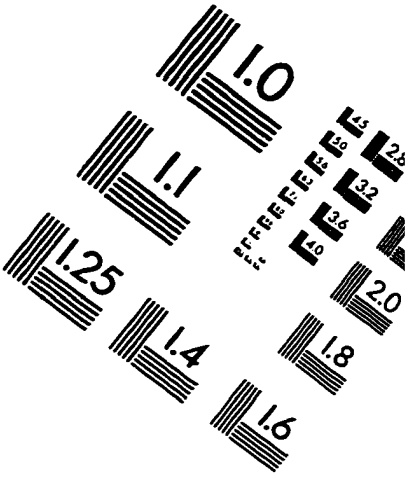
25. Ajay D. Kshemkalyani, Mukesh Singhal, Invarient-Based Verification of a Distributed Deadlock Detection Algorithm, IEEE Transactions on Software Engineering, Vol. 17, NO. 8, August 1991, 789-798

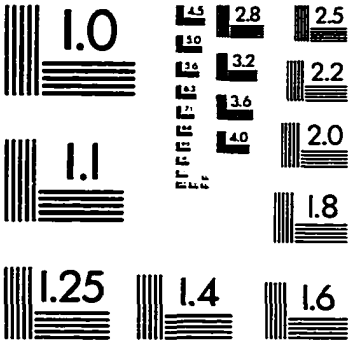
26. Mukesh Singhal, NiranJan G. Shivaratri, Advanced Concepts in Operating Systems, McGraw-Hill, Inc., New York, 1994

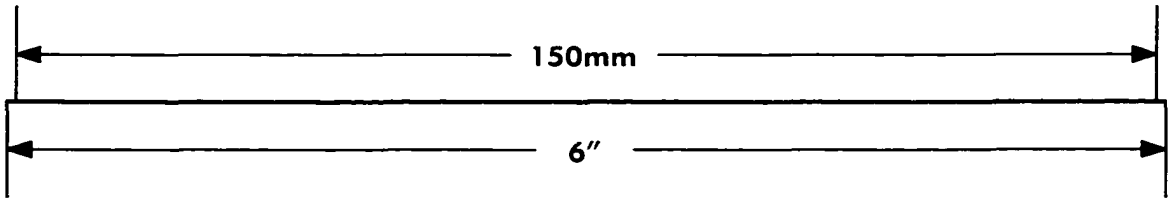
27. Ajay D. Kshemkalyani, Mukesh Sighal, On Characters and Correctness of Distributed Deadlock Detection, Journal of Parallel and Distributed Computing 22, 44-59, 1994

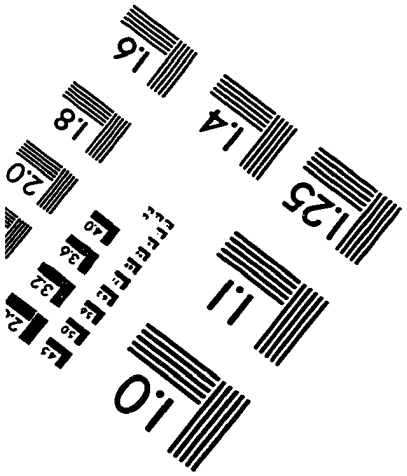
IMAGE EVALUATION TEST TARGET (QA-3)

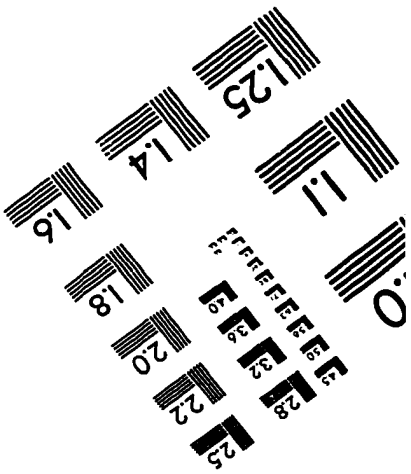












APPLIED IMAGE, Inc
1653 East Main Street
Rochester, NY 14609 USA
Phone: 716/482-0300
Fax: 716/288-5989