

# **Avoiding Deadlock in a Multi-Processor Environment**

**Beth Davis**  
**University of Alabama in Huntsville**  
**Honors Senior Project**  
**Spring Semester 2003**

## **Table of Contents**

**I. Abstract**

**II. Introduction**

**III. Methodology**

**IV. Results**

**V. Discussion**

**VI. Conclusion**

**VII. Acknowledgment**

**VIII. References**

**IX. Graphics**

**X. Appendix A: Algorithms**

**XI. Appendix B: Source Code**

## **Abstract**

Concurrent processing is the means by which a computer can handle multiple tasks at once. Without concurrency, we would not have modern operating systems such as Windows. If one problem could be labeled the scourge of concurrent processing, it would have to be deadlock. Deadlock occurs when two or more processes in a computer system have conflicting needs for resources. While there are methods to predict and prevent deadlock, most modern operating systems do not implement them because the methods require a large amount of resources. The vast majority of operating systems instead focus on detecting deadlock once it has already occurred and then employing some recovery operation. My objective is to find a means of implementing a deadlock avoidance algorithm that is more efficient than the traditional approach of deadlock detection and recovery.

Hypothesis: A multiprocessor environment can be used to implement a deadlock avoidance algorithm.

Question 1: Can a deadlock avoidance algorithm be implemented on a multiprocessor system successfully? *Note: All of the following questions are only valid if the previous question is answered 'yes'.*

Question 2: Will the deadlock avoidance implementation prove to be more efficient in handling deadlock than that of a uniprocessor environment that uses deadlock detection

and recovery?

**Question 3: Is the improvement in efficiency worth the effort to adapt an operating system to a multiprocessor environment?**

Using the instructional operating system known as Nachos, I will simulate a multiprocessor environment. One virtual processor will control the operation of the simulated system and perform its data processing functions. The other processor will run the deadlock avoidance algorithm to determine if any of the processes on the first processor are in danger of deadlock. In addition to analysis of the simulation results and answers to the hypothesis and questions posed above, I will address the meaning and implication(s) of terms such as successfully [Q1], more efficient [Q2], and worth the effort [Q3] in the research paper that will accompany my results.

## **Introduction**

As mentioned in the project abstract, deadlock is a foe to be reckoned with in computing.

There are three typical strategies commonly used to deal with deadlock:

- deadlock prevention
- deadlock avoidance
- deadlock detection and recovery

The former two involve making sure deadlock never occurs, while the latter prescribes solutions for dealing with the mess that deadlock causes. To adopt a strategy that does not offer a solution to the problem is irresponsible; one cannot risk letting deadlock occur in a vital system. For example, the computer system that controls the pulse monitors for patients in a hospital ICU cannot be allowed to fail. Thus, a modern operating system that runs on a mission-critical computer cannot implement a deadlock recovery strategy. Of the two remaining choices, one must decide on the lesser of two evils. Implementing a deadlock prevention strategy means that deadlock is guaranteed to never occur, but the cost is that processes that are "resource - intensive" risk being denied the resources needed to run to completion. Indeed, a man will never stub a toe if he never walks. Therefore, deadlock prevention is too restrictive to be practical. This leaves us with deadlock avoidance. Like deadlock prevention, this strategy also guarantees that deadlock will never happen. The downside is that it consumes a great deal of resources, thus making the system less efficient. This makes deadlock avoidance impractical, but not more so than deadlock prevention. If a deadlock avoidance strategy that does not consume an

unreasonable amount of resources can be implemented, then that strategy would certainly be worthy of more examination. It is this strategy that I wish to explore in this project.

Much effort has been devoted to developing an efficient means of solving the problem of deadlock avoidance. The most popular deadlock avoidance strategy is the Banker's Algorithm, a strategy attributed to the famous computer scientist, Edgar W. Dijkstra [1]. The Banker's Algorithm is used to determine the 'state' of the system. The state is simply a means of referring to the status of the current resource allocations. A 'safe state' means that there is a way of satisfying the requests at hand without throwing the system into deadlock. An 'unsafe state' is the converse of the above; namely, there is no way of satisfying the requests at hand without throwing the system into deadlock. The Banker's Algorithm determines whether a request can be granted safely [2]. For a representation of the Banker's Algorithm in pseudo-code, please refer to Appendix A.

The Banker's Algorithm itself is virtually flawless; it is the implementation of it that is impractical. Every time a request for more resources is made, this algorithm must run its course. For a computer that handles millions of requests in the blink of an eye, this added burden is too much to bear. With this in mind, it seems that the search for a more efficient deadlock avoidance algorithm is rather futile; that is, no matter how quickly the algorithm can be executed, it will still slow down the system tremendously if it must be executed over and over. For a graphical representation of this concept, please refer to Graphic #3. If the problem cannot be solved by devising faster deadlock avoidance algorithms, then a

different plan of attack is needed. If we look at current research in computer science, we see that the problem of deadlock avoidance is being explored in the areas of networks and distributed computing. It seems that the overhead of running the deadlock avoidance algorithm(s) is dealt with by splitting up the work among the different nodes in a network or cluster. This concept can be applied towards a solution for deadlock avoidance in a multi-tasking operating system for a desktop computer. Desktop computers can be made with multiple CPUs; if we think of the CPUs as nodes in a network or cluster, then the distributed solutions for deadlock avoidance can be applied to an operating system. The work involved with running the Banker's Algorithm can be assigned to a processor while another processor handles the other processes for the system. This is the proposed solution for the problem addressed in this project.

## Methodology

The objective of this project is to find a means of implementing a deadlock avoidance algorithm in a multiprocessor environment. The functionality of the program constructed to meet this objective is twofold: 1) the program must implement a deadlock avoidance algorithm, and 2) it must simulate a multiprocessor environment. Before proceeding further, let me address the need to simulate a multiprocessor environment. I do not have access to a computer with multiple processors. Any program I create to run on multiple processors would not function correctly on the computers to which I have access. Therefore, the multiprocessor system must be modeled. To do this, I am drawing on knowledge of the parallels in the structure of multiprocessors systems and that of an abstract representation of processes connected by a form of Inter - Process Communication.

The deadlock avoidance algorithm used here will be the Banker's Algorithm, since it is widely recognized as the definitive answer to deadlock avoidance. The modeling of the multiprocessor system shall be done by having two programs, one to represent each processor in this theoretical system. The first program shall be a modified version of the Nachos operating system. Nachos is a teaching tool used in universities across the country to teach basic and advanced operating system concepts to undergraduates and graduates alike. It simulates the behavior of a modern operating system. Using make a sub-program that submits requests for resources to Nachos, I can then modify Nachos to



send a request to a second program to determine if the requests can safely be granted. This can be accomplished by altering the process initialization code in Nachos to open a line of communication to the second program, which will be waiting for input, and sending a stream of data containing the request information and the current resource status. The second program will run the Banker's Algorithm on the data to determine if the request can be granted and send the result of that deliberation to the first program, which is waiting for a response. The first program will take the response and act according to the recommendation of the second program. That is, if it receives a 'yes' response, it will grant the request; if it receives a 'no' response, it will cause the request to be stored away to be processed again at a later time.

To examine the questions of efficiency and effort, a second simulation must be created for purposes of comparison. This program will be given the same inputs as the first one, and its outputs and response time will be recorded. The program will simply be a slightly modified version of Nachos. Instead of attempting to determine if granting a request is feasible, the program will grant all requests, then terminate operation when a request bankrupts the system. An error message will be displayed to the screen that reports what request dealt the fatal blow.

There will be a minimum of twenty trials for each of these setups. In each case, the conditions will be identical for each simulation. The conditions that will vary from trial to trial are the number of requests and the size of requests. Of the twenty trials, ten will be

purposefully constructed to threaten the system with deadlock. Each simulation will be evaluated according to the following criteria:

- Speed with which all requests are completed. (Calculated by dividing length of time of the simulation by the number of processes completed.)
- Accuracy of request grants (i.e. Did the simulation grant an incorrect amount of resources to a process?)

## **Results**

This project began with the ambition of simulating a multi-processor environment by using a version of the operating system simulation software Nachos that could handle Inter - Process Communication (IPC). In the end, Nachos was not a good choice for this project. The computer I used to work on this project ran SuSE 8.1 Linux as its operating system, one that used the pthreads library<sup>1</sup>. Nachos was written for a Sun Solaris-type platform<sup>2</sup>, which means that almost every source file had references to a proprietary thread library that didn't exist on my computer. The work involved in converting the Nachos source code into something useable on my computer was too time-consuming; a change in plans was necessary.

'Plan B' consisted of making a home-grown version of what I needed Nachos to do and then proceeding as originally planned. All I really needed from Nachos was a way of simulating what happens when a process submits a request for resources to the processor. I did this by reading the request information from a file into a multi-dimensional array, then feeding each row of this matrix to a thread. Each row from this matrix represented a new request for resources. The threads represented the processes. To represent the resources of

---

<sup>1</sup> A library is a collection of code that is made available for other programmers to use. Using libraries allows programmers to quickly compose programs without having to reinvent the wheel, so to speak. The pthreads library is a collection of code that lets a programmer create, use and destroy POSIX standard threads.

<sup>2</sup> Solaris is a type of computer made by Sun Microsystems. It also refers to the particular type of the Unix operating system that runs on these computers.

a system, I made a globally-accessible array of the same dimension as the request vectors. Each thread attempted to fill its request by subtracting each element in the request vector from the corresponding element in the global resource vector. If the value of any element in the global resource vector dipped below 0, an error message displayed to the screen. In real life, this event would be described as a particular form of deadlock. It would probably be characterized by the infamous "Blue Screen of Death" in a Win32 environment<sup>3</sup>, or an "insufficient memory" error in most environments. Please see Appendix B for the source code.

The program described above was intended to use to demonstrate what deadlock is; another program was needed to test my hypothesis that a deadlock avoidance algorithm could be implemented in a multi-processor environment. To do this, I needed to modify the program to serve two new functions: 1) simulate a multi-processor environment and 2) implement a deadlock avoidance algorithm. It is important to note that I could only try to *simulate* a multi-processor environment, as the only computers to which I had access were single-processor systems. The simulation was set up by using two programs that could communicate with each other. One program would be my original deadlock simulator. The other would be an implementation of Dijkstra's Banker's Algorithm, the authoritative method for predicting an unsafe state.

---

<sup>3</sup> Win32 is a popular way of referring to a Windows operating system without having to specify whether it is Windows 3.1, 3.11, 95, 98, etc.

The original program would be modified so that instead of giving an error message after the deadlock occurred, it would first try to find out if the granting of resources would result in an unsafe state. If the state was determined to be safe, then the allocation of resources would proceed as normal. If the state was unsafe, the thread was to go into a waiting mode, so that other threads would have time to replace the resources needed to continue with the allocation of resources. The actual determination of a safe or unsafe state would take place in the second program, so to get that information, the first program would need to communicate with the second program. This would be accomplished by opening a socket<sup>4</sup> and sending the necessary information (current resource vector and request vector) to the second program. Once the second program had performed its calculation, it would use the socket to relay its results back to the first program.

If the first program was considered a 'client', then the second program was the 'server'. Once the second program was initiated, it was to wait passively for the first program to send it a request. When it did so, the second program would look at the data provided and determine if the request could be satisfied without bankrupting the resource vector. Again, the Banker's Algorithm would be the core of this program. The program would make its calculation and encode its result as either a 0 or a 1, with 0 meaning "safe" and 1 meaning "unsafe". After the algorithm finished, the result would then be sent to the first program via a socket.

---

<sup>4</sup> A socket is a programming construct used to communicate with other processes. It may be helpful to think of opening a socket as picking up a telephone.

These two programs would work together as two processors in a multi-processor environment would, with the socket communications acting as the system bus<sup>5</sup>. The processing of resource requests would be handled by one "processor", while the other "processor" would determine if the resource requests could be satisfied. Unfortunately, the best laid plans of mice and men go oft awry. In the implementation of this plan, the transferral of data between the two programs became an obstacle that could not be overcome in a timely fashion. Though the system cannot be presented in working order, there are lessons to be learned in the process. The Discussion section deals with these and other lessons.

---

<sup>5</sup> A system bus is a means of communication between the hardware components in a computer. For example, data travels from the main memory to the CPU on the bus. If a system has more than one processor, data will flow between the two on the bus.

## **Discussion**

At first blush, this project seemed to have great promise. An efficient solution for deadlock avoidance for desktop computers would be a great boon to computer users. I may continue this line of research in the future, but first a discussion is needed of the problems encountered and their possible solutions.

The first and most glaring problem is that the source code needed to simulate a dual-processor system did not execute properly. This is most likely because of the function parameters I tried to pass through the pthread creation function. When a thread is created, a programmer has to tell it what its identification is, what properties it has, what its task is, and what data it needs to complete that task. The data that my threads needed to complete their respective tasks was given to them in an improper form. I could not resolve this problem quickly, but it is not an insurmountable task. If I had budgeted my time more wisely, then this problem might have been solved.

The second problem was with the communication between the two programs. The second program was expecting a copy of the data from the first program; instead, it was receiving a copy of a reference to the original data. When the server program received its data, it was in an unusable form. Again, this is a problem that could have been overcome if I had allowed more time to deal with setbacks like this.

Another problem with my plan is that the idea of deadlock avoidance only works if the entirety of a system is analyzed. That is, a successful method would entail not only comparing the request to the remaining resources, but also looking at the request in the context of all the other requests in the system. The power of the Banker's Algorithm lies in its ability to predict whether the fulfilling of a request could *possibly* put the system in danger, not whether such danger is imminent. To modify my program to correctly implement the Banker's Algorithm, the client program would have to send the server program a data structure that contained the state of the system, not just the request and resource vectors. Graphic #4 illustrates the basic algorithm behind this project and how it must be modified to successfully utilize the Banker's Algorithm.



## **Conclusion**

As mentioned before, this was a project with promise. With some modifications, it would be a worthwhile pursuit for those so inclined. I say "so inclined" because the person who pursues this project would have to possess knowledge of both hardware design and operating system concepts. An individual who is willing to not only purchase/make a multi-processor computer but also to modify one of the open-source operating systems would find that implementing the solution proscribed here would be worthwhile. The average computer user, however, would not be willing to undertake such a task and would instead wish to purchase a computer system that was already modified. This consumer would have to evaluate such a purchase, of course, by taking into consideration such factors as ease of use, price, functionality, and so forth. A computer system that was modified to implement my solution to the deadlock avoidance problem would probably cost more than the average computer system on the market today. The consumer would have to decide whether the elimination of deadlock is worth the extra cost. The hardware manufacturer would have to determine if producing multi-processor computers is cost-effective. Software companies that create operating systems (i.e., Microsoft) would need to evaluate whether modifying their software would be a reasonable investment of their time and money. In short, this project's commercial success would depend on the willingness of both computer manufacturers and operating system developers to invest their capital, as well as on the demand from consumers.

## **Acknowledgments**

I would like to thank Dr. Glenn Cox for serving as my advisor on this project. His help in shaping the goal of this project was invaluable.

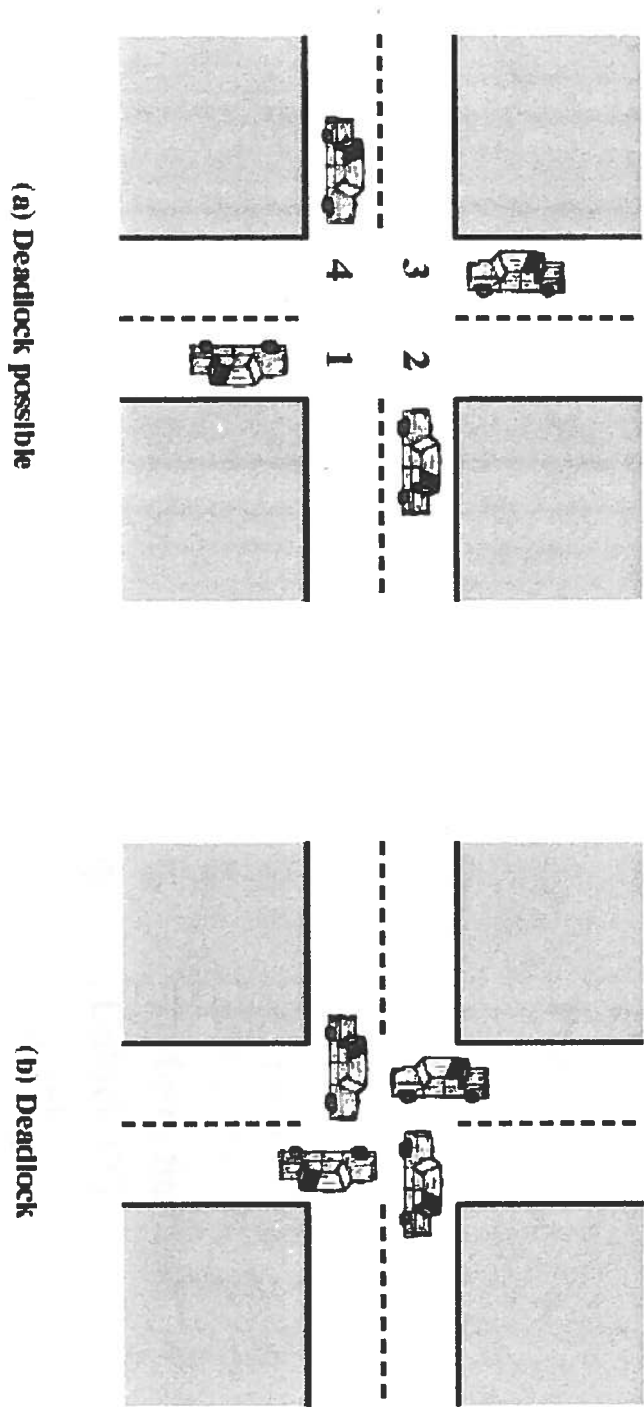
Thanks are also in order for Dr. Jerry Mebane for his leadership and Betty Cole, for her friendship and advice.

Mom, thanks for everything.

## **References**

- [1] Stallings, William. Operating Systems, Fourth Edition. Prentice Hall. 2001.
- [2] Kim, In Guk. New Mexico Institute of Mining and Technology.  
[ftp://ftp.prenhall.com/pub/esm/computer\\_science.s-041/stallings/Slides/OS4e-PDF-Slides](ftp://ftp.prenhall.com/pub/esm/computer_science.s-041/stallings/Slides/OS4e-PDF-Slides)  
Accessed March 22, 2003.

**Graphic #1: Abstract Representation of Deadlock**



**Figure 6.1 Illustration of Deadlock**

Reference [1]

**Graphic #2: Example of Deadlock**

**Process P**

Step	Action
P <sub>0</sub>	Request (D)
P <sub>1</sub>	Lock (D)
P <sub>2</sub>	Request (T)
P <sub>3</sub>	Lock (T)
P <sub>4</sub>	Perform function
P <sub>5</sub>	Unlock (D)
P <sub>6</sub>	Unlock (T)

**Process Q**

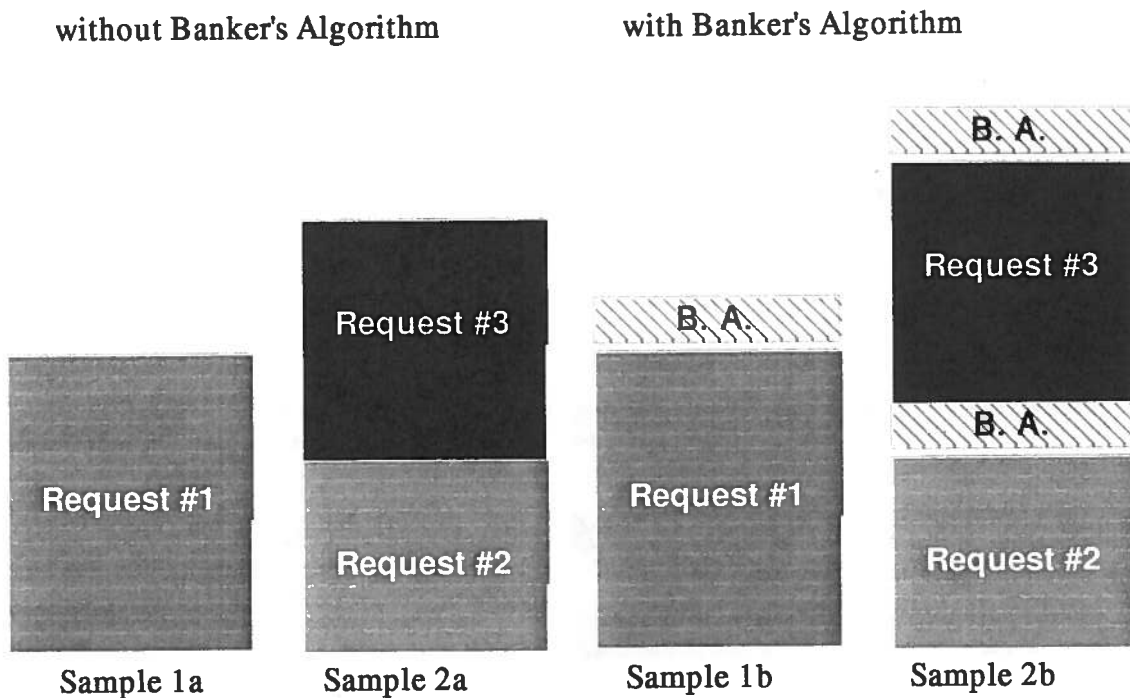
Step	Action
q <sub>0</sub>	Request (T)
q <sub>1</sub>	Lock (T)
q <sub>2</sub>	Request (D)
q <sub>3</sub>	Lock (D)
q <sub>4</sub>	Perform function
q <sub>5</sub>	Unlock (T)
q <sub>6</sub>	Unlock (D)

**Figure 6.4 Example of Two Processes Competing for Reusable Resources**

Reference [1]

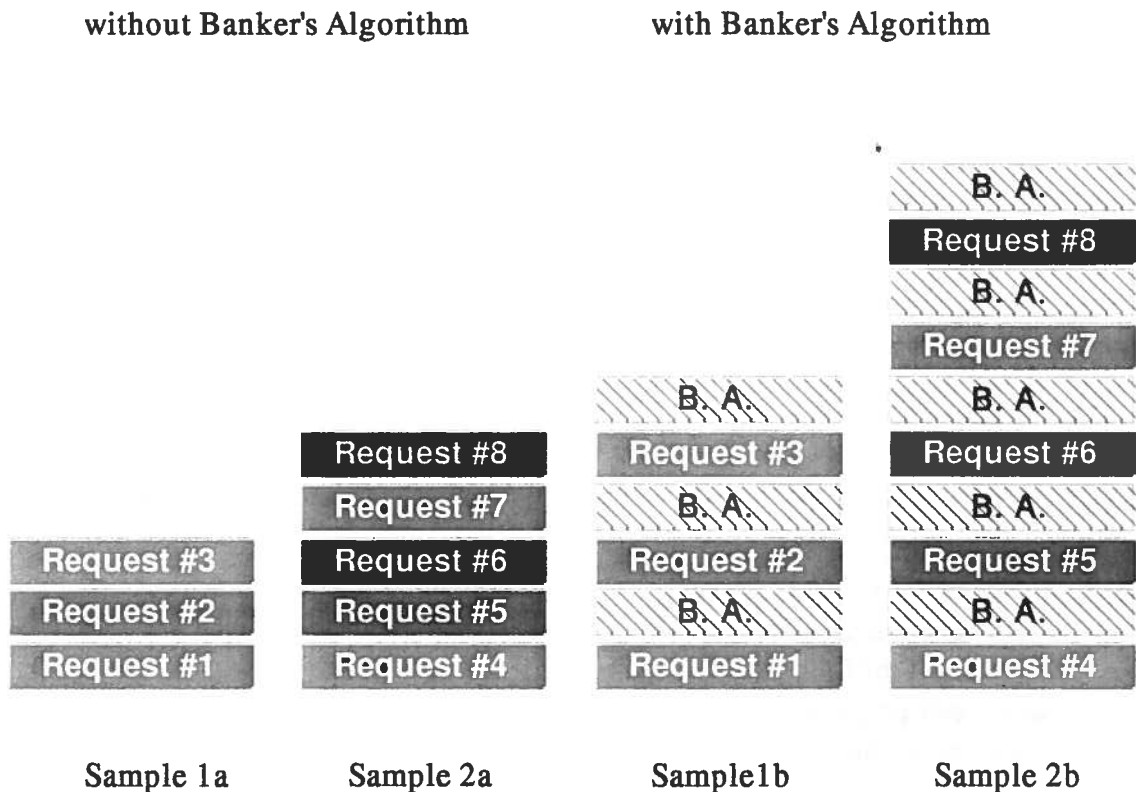
### Graphic #3: Representation of Burden that Banker's Algorithm Places on Computer System

As you can see, running the Banker's Algorithm would slow down a system by a small amount if there are only a few requests made per clock tick.

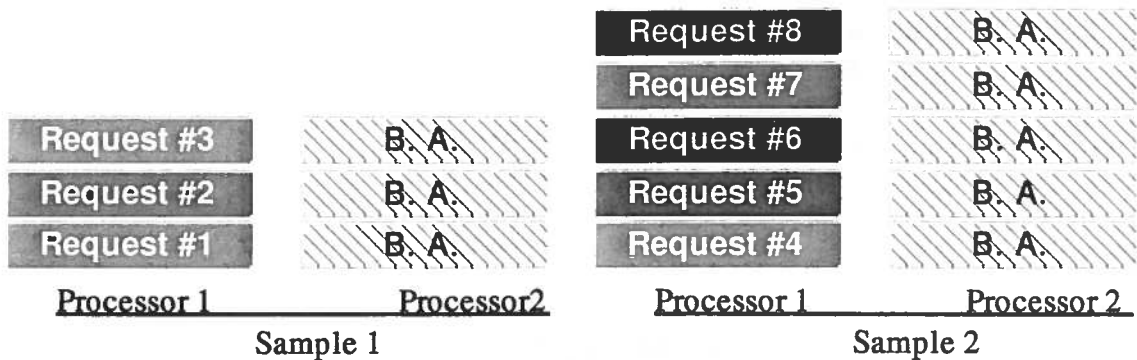


Note: Each block represents the amount of time the processor must spend to process the request. Larger blocks represent processes that take more time. Samples 1a and 1b represent the same request, but 1b includes a block of time to process the Banker's Algorithm. The same is true of samples 2a and 2b.

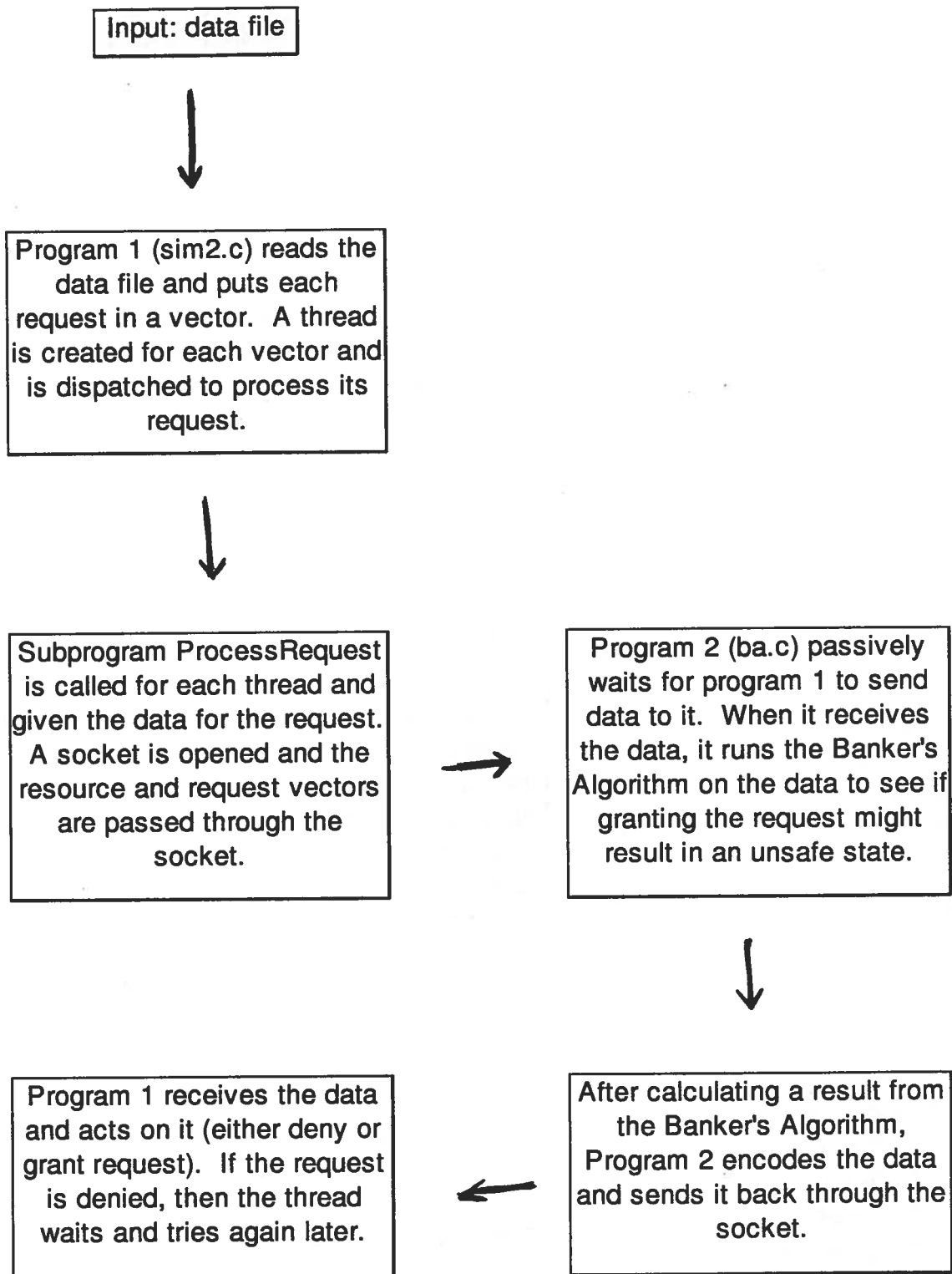
The problem arises when there are a large number of small requests. The processor would spend almost as much time running the Banker's Algorithm as it would for the regular resource requests.



If we could separate the processing of the Banker's Algorithm from the processing of the requests, the representation would look like this:

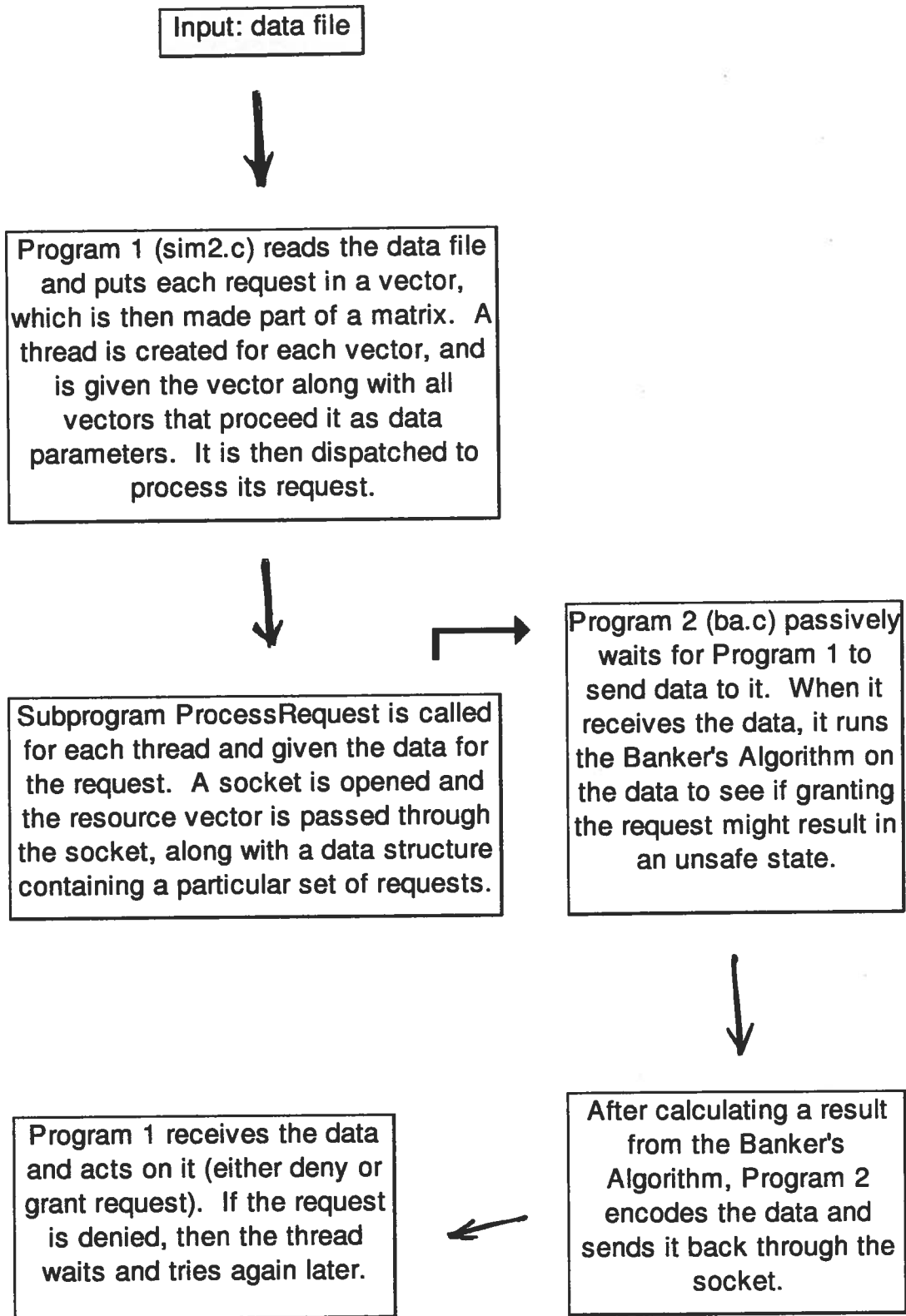


#### Graphic #4: Representation of Original Algorithm and Modified Algorithm



**Original Algorithm (Flowchart)**





**Modified Algorithm (Flowchart)**

## Appendix A: Algorithms

### Banker's Algorithm

```
public class state {
    int [] resource = new int[m];
    int [] available = new int[m];
    int [][] claim = new int[n][m];
    int [][] alloc = new int[n][m];
}

if (alloc [i,*] + request [*] > claim [i, *]){
    <error>;    //--total request > claim
} else if (request [*] > available [*]){
    <suspend process>;
} else {
    //--simulate alloc
    <define newstate by:
    alloc [i, *] = alloc [i, *] + request [*];
    available [*] = available [*] - request [*] >;
}
if (safe (newstate)) {
    <carry out alloc>;
} else {
    <restore original state>;
    <suspend process>;
}

public boolean safe (state S)
{
    int [] currentavail = new int [m];
    process [] rest = new process [<number of processes>];
    currentavail = available;
    rest = { all processes };
    possible = true;
    while (possible) {
        find a Pk in rest such that
            claim [k, *] - alloc [k, *] <= currentavail;
        if (found) {    //simulate execution of P
            currentavail = currentavail + alloc [k, *];
            rest = rest - { Pk };
        }
        else
            possible = false;
    }
    return (rest == null);
}
```

## Appendix B: Source Code Listing

This is the listing for `sim1.c`, the program that simulated deadlock. It does not require any command-line arguments, but it needs a file called 'input.dat' containing a 3 x 4 matrix of integers in order to read in and process requests.

```
/*
*****
****
* FILE: sim1.c
* DESCRIPTION:
*   A program which demonstrates the processing of resource
*   requests. This program simulates a particular form of
*   deadlock.
*
*****
*****/

#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <stddef.h>
#include <ctype.h>

#define NUM_THREADS      3
#define MAXLINE 100
#define MAXROWS 20
#define MAXCOLS 20

/*Global Resource Array*/
int resources[4] = {50, 50, 50, 50};

struct thread_data
{
    int thread_id;
    int req_vector[4];
};

struct thread_data thread_data_array[NUM_THREADS];

/* Read one line from fp, */
/* copying it to line array (but no more than max chars). */
/* Does not place terminating \n in line array. */
/* Returns line length, or 0 for empty line, or EOF for end-of-file.
*/

int fgetline(FILE *fp, char line[], int max)
{
    int nch = 0;
    int c;
    max = max - 1;
    /* leave room for '\0' */

```

```

while((c = getc(fp)) != EOF)
{
    if(c == '\n')
        break;

    if(nch < max)
    {
        line[nch] = c;
        nch = nch + 1;
    }

    if(c == EOF && nch == 0)
        return EOF;

    line[nch] = '\0';
    return nch;
}

/* Parse line into words. Return number of words. */
int getwords(char *line, char *words[], int maxwords)
{
    char *p = line;
    int nwords = 0;

    while(1)
    {
        while(isspace(*p))
            p++;

        if(*p == '\0')
            return nwords;

        words[nwords++] = p;

        while(!isspace(*p) && *p != '\0')
            p++;

        if(*p == '\0')
            return nwords;

        *p++ = '\0';

        if(nwords >= maxwords)
            return nwords;
    }
}

void *ProcessRequest(void *threadarg)
{
    int taskid, i;
    int my_vector[4];

```

```

struct thread_data *my_data;

sleep(1);
my_data = (struct thread_data *) threadarg;
taskid = my_data->thread_id;

printf("Thread %d: \n", taskid);
printf("Request vector is :");
for (i = 0; i < 4; i ++)
{
    my_vector[i] = my_data->req_vector[i];
    printf("%d\t", my_vector[i]);
}
printf("\n");
for (i = 0; i < 4; i++)
{
    resources[i] = resources[i] - my_vector[i];
    if (resources[i] < 0)
    {
        printf("Emergency! Resource # %d is depleted.\n", i);
        pthread_exit(NULL);
        exit(0);
    }
}
pthread_exit(NULL);
}

void *PutItBack(void *threadarg)
{
    int taskid, i;
    int my_vector[4];
    struct thread_data *my_data;

    sleep(1);
    my_data = (struct thread_data *) threadarg;
    taskid = my_data->thread_id;

    printf("Thread %d: \n", taskid);
    printf("Replacing resources :");
    for (i = 0; i < 4; i ++)
    {
        my_vector[i] = my_data->req_vector[i];
        printf("%d\t", my_vector[i]);
    }
    printf("\n");
    for (i = 0; i < 4; i++)
    {
        resources[i] = resources[i] + my_vector[i];
    }
    pthread_exit(NULL);
}

int main(int argc, char *argv[])
{

```

```

pthread_t threads[NUM_THREADS];
int *taskids[NUM_THREADS];
int rc, t;

    /*variables for reading in and processing request matrix*/
int array[MAXROWS][MAXCOLS];
int request[3][4];
char *filename = "input.dat";
FILE *ifp;
char line[MAXLINE];
char *words[MAXCOLS];
int nrows = 0;
int n, i, j, k;

/*Read in request matrix*/
ifp = fopen(filename, "r");
if(ifp == NULL)
{
    fprintf(stderr, "can't open %s\n", filename);
    exit(EXIT_FAILURE);
}
k = 0;

while(fgetline(ifp, line, MAXLINE) != EOF)
{
    if(nrows >= MAXROWS)
    {
        fprintf(stderr, "too many rows\n");
        exit(EXIT_FAILURE);
    }

    n = getwords(line, words, MAXCOLS);

    for(i = 0; i < n; i++)
    {
        array[nrows][i] = atoi(words[i]);
        //printf("%d\t", array[nrows][i]);
        request[k][i] = array[nrows][i];
        nrows++;
    }

    //printf("\n");
    k++;
}

printf("Contents of request matrix:\n");

for (i = 0; i < 3; i++)
{
    for (j = 0; j < 4; j++)
    {
        printf("%d\t", request[i][j]);
    }
}

```

```

    printf("\n");
}

printf("\n\n");

/*Generate threads to process requests*/
for(t=0;t<NUM_THREADS;t++)
{
    thread_data_array[t].thread_id = t;

    for (j = 0; j < 4; j++)
    {
        thread_data_array[t].req_vector[j] = request[t][j];
    }
    printf("Creating thread %d\n", t);
    rc = pthread_create(&threads[t], NULL, ProcessRequest, (void *)
&thread_data_array[t]);
    if (rc)
    {
        printf("ERROR; return code from pthread_create() is %d\n",
rc);
        exit(-1);
    }
} //End for loop

/*Generate threads to replace resources*/
for(t=0;t<NUM_THREADS;t++)
{
    thread_data_array[t].thread_id = t;

    for (j = 0; j < 4; j++)
    {
        thread_data_array[t].req_vector[j] = request[t][j];
    }
    printf("Creating thread %d\n", t);
    rc = pthread_create(&threads[t], NULL, PutItBack, (void *)
&thread_data_array[t]);
    if (rc)
    {
        printf("ERROR; return code from pthread_create() is %d\n",
rc);
        exit(-1);
    }
} //End for loop
pthread_exit(NULL);
} //end main

```

**Honors Senior Project  
Approval**

Form 3 – Submit with completed thesis. All signatures must be obtained.

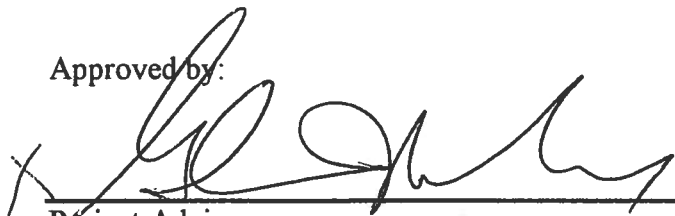
Name of candidate: Beth Davis

Department: Computer Science

Degree: B.S. - Computer Science

Full title of project: Avoiding Deadlock in a Multi-Processor  
Environment

Approved by:

X 	<u>6-16-03</u>
Project Advisor	Date
X <u>Heggere S. Ranyavath</u>	<u>6-16-03</u>
Department Chair	Date
<u>Jeenu L. Mehare</u>	<u>6-19-03</u>
Honors Program Director for Honors Council	Date