Depth-first Search Algorithm Implementation on Detecting Possible Deadlocks on Operating Systems

Muhammad Fauzan Rafi Sidiq Widjonarto - 13518147
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
fauzanrafisidiq@gmail.com 13518147@std.stei.itb.ac.id

***Abstract*—Operating systems are inseparable in the modern life, embedded in various Internet of Things and computers of various sizes. They are used mainly for providing useful interface to users, but one of the services the operating system provide is resource allocation for processes. Upon these allocation processes, a state called a deadlocked state can arise, where all process involved is waiting for another forever, thus crashing the operating system. Operating systems need to have a deadlock avoidance mechanism to know whether the current state will result in deadlocked state. In this paper, author will discuss the implementation of depth-first search algorithm to a graph modelled state of process to determine whether it will resulted in a deadlock or not.**

Keywords—depth-first search; deadlock; operating system; resource allocation; graph modelling;

I. Introduction

Operating systems are the organizers of computers, handling almost every aspect possible about a unit of computer. The need of operating systems has arise, where all kinds of devices and services has been embedded with some level of digitalization need a form of operating system. Trends in society about the Internet of Things (IoT) and microcomputers have arise because of the repeated Fourth Industrial Revolution narration. In effect, markets for various operating systems for these devices have arise as well [1].

One of operating system's service is the allocation of resources to existing processes. Not only to ensure the process' success, but also to synchronize processes of mutual resource needs. This service is important for running various processes in a computer, defining its usability in the real world. Thus, this service and usage is instrumental in any computers.

According to Silberschatz [2], in a multiprogramming environment, several processes may compete for a finite number of resources. A process requests resources; if the resources are not available at that time, the process enters a waiting state. Sometimes, a waiting process is never again able to change state, because the resources it has requested are held by other waiting processes. This situation is called a deadlock.

To prevent this, operating systems have mechanisms to avoid deadlock. One of them is to determine whether a state is may result in deadlock or not from the current state of process and resources. Provided these information, it is possible to model the state as a graph, and from the model the implementation of depth-first search algorithm to solve this problem will be discussed further in this paper.

II. Theoretical Framework

A. Graph

A graph is a discrete mathematical structure in which every object defined on the graph is related to one another in some configuration. The objects of the mathematical abstraction corresponds in a graph is called vertices. The representation of the connections of the objects, which is the connection between the vertices, is called edges. Degree is the amount of edges adjacent to a vertex.

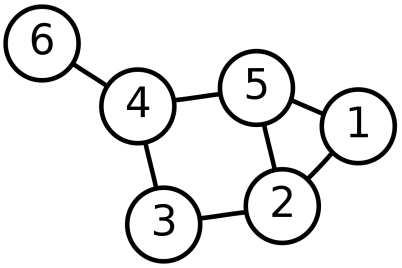

Figure 1. Source: https://en.wikipedia.org/wiki/Graph_(discrete_mathematics)

For example, in figure 1, node 6 has a degree of 1, while node 5 has a degree of 3. Two nodes are said to be adjacent if they're connected with the same edge. Node 6 and node 4 are adjacent to each other, but node 6 and node 1 aren't adjacent to each other.

Graph has many variations, but for modelling interactions of objects there are three useful variations of graphs:

1. Undirected Graph, where the edges of the graph don't have any direction in the connection.
2. Directed Graph, where the edges of the graph have direction in the connection.

3. Weighted Graph, where the graphs edges have a number (the weight) is assigned to each edge. Such weights might represent for example costs, lengths or capacities.

Graph has been considered as one of the best mathematical models because of its applicability in almost any problems. It is also widely used as models because of vast amount of "connectivity" problems in the modern world. Not just in mathematics, but also in computer science to model networks, sociology to model social circles and social networks, and in network theory as a fundamental concept.

B. Resource Allocation on Operating Systems

On operating systems, resource allocation has been one of the instrumental services that is important to be done right. If not, then the underlying management of the operating system is at best a failure. One of the reasons why operating systems came to being is to organize and allocate resources available for best use [2]. In that case, operating systems must have tools to allocate available resources to be assigned to existing processes.

One of ways to model processes trying to get resources is using a graph. This model is called the Resource Allocation Graph, illustrated in figure 2. This special graph has these properties:

1. It has two types of nodes: one representing a process and one representing a resource
2. The graph is a directed graph, where every nodes always alternates in connectivity: a process node always connect to resource node, and vice versa.
3. A resource nodes degree indicates its instance on the underlying system: if it has a degree of one, then the resource is instantiated exactly once, etc.
4. The direction of the edges has a special meaning: if it is directed from a process node then it means that the process wants to acquire a resource. If it is directed towards a process node the it means that the resource is already been acquired by the process

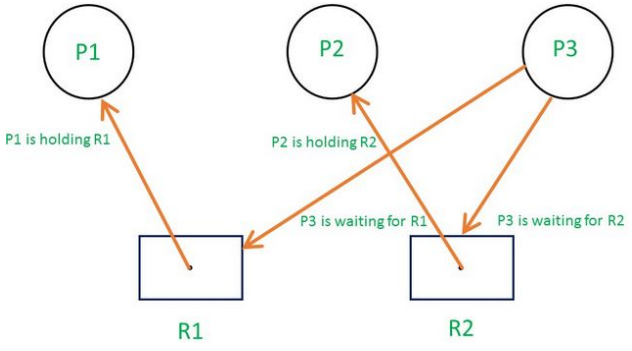

Figure 2. Source: https://www.geeksforgeeks.org/resource-allocation-graph-rag-in-operating-system/

When two or more process is trying to acquire an acquired resource of one another, a state called deadlock has occured. The processes involved cannot run or activated because it waits for another, forever. Thus, at this point the program can crash. With the graph model, a deadlock can be detected if there is a cycle on the Resource Allocation Graph model. If the model is a single instance resource model, then a cycle in the system means the system is in a deadlocked state. But in a multi instance model of the graph, a cycle means that a deadlock may be occured, but requires further examination. Thus, the deadlock prevention methodologies of the operating system must prevent a cycle to be formed, or it must assess the state to analyze a possible deadlock in the future.

C. Depth-first Search Algorithm

Depth-first search (DFS) algorithm is a graph-traversing algorithm where the traversing order starts from "depth" first, traversing on every child until it cannot be expanded anymore and starts to backtrack, until all of the nodes connected are visited.

DFS is used for many purposes and exists in many different types to serve the purpose of the problem. But the DFS algorithm has a general shape of algorithm, as the following:

1. Initialize an array to indicate that the i-th node has been visited. Let it be called visited
2. Traverse the graph from the initial node. Process the node for the purpose of the problem, and then mark the node visited. (visited[*init_node*] = true).
3. If the node doesn't have any adjacent nodes, then backtrack to the parent node. Backtracking means find all remaining unvisited nodes.
4. If the node in fact have adjacent nodes, then repeat step 2 for all of the adjacent nodes.
5. If all nodes have been visited, then stop. The traversing is done.

The algorithm of DFS can be realized using recursion or using stack data structure. The implementation of the algorithm is not bounded to one form or another.

DFS algorithm is widely used in detecting cycles on a graph due to its nature of traversing through the depths of the graph, and finding the strongly connected parts of a graph, that is two or more nodes of a graph connected to one another, creating a cycle. And we will see that the DFS algorithm will prove to be a good solution on solving cycle problem, in which our deadlock problem is modelled.

III. Implementation

In this paper, the state of process and resources is modelled using a graph, more specifically Resource Allocation Graph. The program is made to detect any cycles on the graph model using DFS algorithm. If a cycle is detected, then the current state may turn into a deadlocked state, so the programs output will be a "warning" of a possible upcoming deadlock state. But if no cycle is detected, then the program return a safe message.

The program is written in C++ language. The program's input is an external file of a state table of the process and resources with the following format:

```
<number of resource>
<all resource names>
<number of process>
(repeat for all process)
<i-th process>
<i-th process acquired resource>
<resources i-th process wants to acquire>
(end repeat)
```

A file example:

example.txt

```
5
A B C D E
2
P1
1 A
1 B
P2
3 A B C
2 D E
```

The example means that there are five resources in total, labeled A through E. And then there are two processes, P1 and P2. P1 already acquired resource A and trying to acquire B. P2 already acquired A, B, C and trying to acquire D and E. Note that it is implicitly stated that the model is a multi instance model because A is already acquired by more than one processes.

To model the graph, we use a modified data structure using C++ standard library and built-in data structures:

```
typedef struct {
    vector<string> resourceNodes;
    vector<string> processNodes;
    vector<pair<string, string>> connectivity;
} Graph;
```

Here are the explanation of the struct fields:

- The graph is modelled based on the multi-instance model (set to be true). If it is in fact modelled using single instance, this field is set to be false. But
- `vector<string> resourceNodes` is a field to set track on every resources on the graph.
- `vector<string> processNodes` is a field to set track on every process on the graph.
- `vector<pair<string, string>> connectivity` is a field to map all connectivity of the nodes to a vector of pairs: where the directed graph is directed from the first element of the tuple.

Program for parsing the external file is as following:

```
#include <bits/stdc++.h>
using namespace std;

void parseToGraph(Graph * g, string filename) {
    ifstream file;
    string hold;
    int n;
    file.open("example.txt");
    /* Read number of resources */
    file >> n;

    /* Plug the resources to graph */
    for(int i = 0; i < n; i++) {
        file >> hold;
            g->resourceNodes.push_back(hold);
    }

    /* Read number of processes */
    file >> n;

    /* Plug the resources to graph, and make
connectivity array */
    for(int i = 0; i < n; i++) {
        string pName;
        int size;
        file >> pName;

        g->processNodes.push_back(pName);

        file >> size;

        for(int j = 0; j < size; j++) {
            file >> hold;
        pair <string, string> p = make_pair(hold,
pName);
            g->connectivity.push_back(p);
        }

        file >> size;

        for(int j = 0; j < size; j++) {
            file >> hold;
            pair <string, string> p =
make_pair(pName, hold);
            g->connectivity.push_back(p);
        }
    }

    for(auto res : g->resourceNodes) {
            g->visited[res] = false;
    }

    for(auto pro : g->processNodes) {
            g->visited[pro] = false;
```

```
    }

    file.close();
}
```

Finally, the algorithm to detect cycles on the graph with DFS algorithm is as follows:

```
void resetVisited(Graph * g) {
    for(auto a : g->visited) {
        a.second = false;
    }
}

bool isCycleExistFromNode(Graph * g, string node,
vector<string> parents) {
    vector<string> adjacent;
    int cnt = 0;
    bool returningStatus = false;
    bool searchParent = false;

    for(auto a : parents) {
        if(a == node) {
            searchParent = true;
            break;
        }
    }

    if(g->visited[node] || searchParent) {
            return true;
    }

    g->visited[node] = true;
    parents.push_back(node);

    for(auto n : g->connectivity) {
        if(n.first == node) {
            adjacent.push_back(n.second);
        }
    }

    if(adjacent.size() > 0) {
        for(auto n : adjacent) {
            returningStatus = returningStatus ||
isCycleExistFromNode(g, n, parents);
        }
    }

    return returningStatus;
}

bool isCycleExist(Graph g) {
    /* Keep track paths from the recursion */
    vector<string> parents;

    for(int i = 0; i < g.processNodes.size(); i++)
{
        resetVisited(&g);
        if(isCycleExistFromNode(&g,
g.processNodes[i], parents)) {
            return true;
        }
    }
    return false;
}
```

The DFS algorithm utilizes function to detect a cycle from every node to ensure every possible cycles, due to the fact that the graph model is a directed graph. The function starts from every node of the graph to ensure all cycles are covered and detected. The DFS algorithm finds a cycle if an expanded node is already visited before as the parent of itself, thus a cycle is detected. If this happens, then the program returns true. If all of the nodes are already visited, then no cycles have been detected and returns false.

The driver code for the program is as follows:

```
int main(int argc, char* argv[]) {
    /* Initialize variables */
    Graph RAG;
    string filename;

    /* Error message for file error */
    if(argc < 1) {
        cerr << "No file submitted to be
analyzed";
        return -1;
    }

    /* input filename from command */
    filename = argv[1];

    /* Parse external file to graph data structure
*/
    parseToGraph(&RAG, filename);

    /* Analyze using DFS is cycle exist on graph
*/
    if(isCycleExist(RAG)) {
            cout << "The state is NOT safe. " <<
        "The state of Deadlock may arise from this
state." <<
        endl;
    } else {
            cout << "The state is safe." << endl;
    }
```

```
    return 0;
}
```

The program receives the filename from the command line receiver, hence the use of arguments on the main driver function. It is signals the possible deadlock state from the DFS analysis.

IV. Case Studies

In this chapter, the cycle-finding algorithm will be executed on a few case study. Specifically on cases of disjoint graph models, cyclic multi-instance and single-instance model, and safe states of the processes.

First test is the example on the previous chapter, renamed as tc1.txt:

tc1.txt

```
5
A B C D E
2
P1
1 A
1 B
P2
3 A B C
2 D E
```

Here, the state is safe because the resources are multi-instanced and the process P2 is gaining the unallocated resource, hence can be done and releasing B in effect.

The result is as follows:

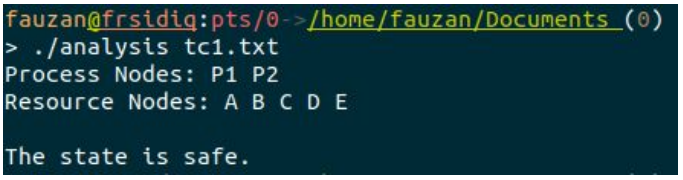

Figure 3

Second test is the example of cyclic single-instanced state, named tc2.txt:

tc2.txt

```
3
A B C
3
P1
1 A
1 B
P2
1 B
1 C
P3
1 C
1 A
```

Here, the state is not safe due to the circular wait: a state where of waiting processes must exist such that P1 is waiting for a resource held by P2 , P2 is waiting for a resource held by P3, and P3 is waiting for resource held by P1.

The result is as follows:

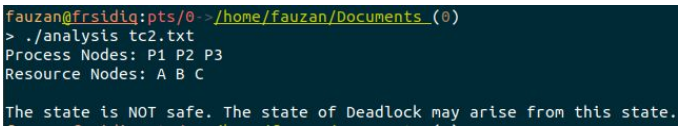

Figure 4

Third test is the example of cyclic single-instanced state with disjoint graph model, named tc3.txt:

tc3.txt

```
4
A B C D
3
P1
1 A
1 B
P2
1 B
1 A
P3
1 C
1 D
P4
1 D
1 C
```

Here, the state is not safe due to circular waits on different graph: a disjoint graph, due to processes needing different resources:

The result is as follows:

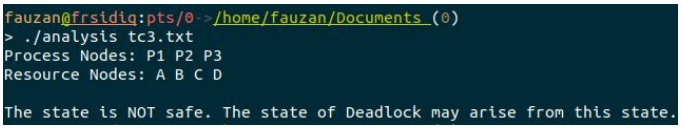

Figure 5

Fourth test is the example normal disjoint graphs, renamed as tc4.txt:

tc4.txt

```
8
A B C D E F G H
4
P1
1 A
1 B
P2
```

```
3 A B C
2 D E
P3
1 F
1 G
P4
1 G
1 H
```

Here, the state is safe, similar to tc1.txt, with addition on the second disjoint graph: process P4 is gaining the unallocated resource, hence can be done and releasing G in effect to process P3, finishing the process in effect.

The result is as follows:

```
fauzan@frsidiq:pts/0->/home/fauzan/Documents (0)
> ./analysis tc4.txt
Process Nodes: P1 P2 P3 P4
Resource Nodes: A B C D E F G H

The state is safe.
```

Figure 6

This warning system using DFS to detect possible deadlocks can be integrated with the underlying system of the operating system. If a deadlock in fact detected, then the system can perform much more sophisticated methods to detect the deadlock, such as Banker's Algorithm.

Overall, the performance of the algorithm is as the following table:

Test Case	**Result**
Normal state	Correct
Circular wait state	Correct
Disjoint cyclic state	Correct
Disjoint normal state	Correct

Table 1. Result of the program with various test cases

The tests of the case studies are performed with machine with this specification:

- OS: Ubuntu 18.04.4 LTS x86_64
- CPU: Intel i5-8250U (8) @ 3.400GHz
- GPU: NVIDIA GeForce MX150
- Memory: 3951MiB / 7869MiB

V. Conclusion

Depth-first search algorithm can be used to detect cycles on a model graph, and prove to be very useful to solve the detection of possible deadlock state in process and resource management problem of operating systems, thus can improve the performance of the operating system and improving the deadlock avoidance techniques.

Source Code Link

https://github.com/mufraswid/dfs-detecting-deadlock

Video Link at Youtube

https://youtu.be/KcD4l_W0_Pk

Acknowledgment

The author would like to thank God the Almighty for His grace and blessings. The author would also thank Dr. Ir. Rinaldi Munir, M.T., as the lecturer of Algorithm Design (IF2211), the author's family, and friends for their support in the making of this paper.

References

[1] https://www.cisco.com/c/en/us/solutions/internet-of-things/future-of-iot.html

[2] A. Silberschatz, P.B Gavin, and G. Gagne, "Operating System Concepts" Phil. Trans. Roy. Soc. London, 8th Edition.

PERNYATAAN

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 3 Mei 2020

Muhammad Fauzan Rafi Sidiq Widjonarto 13518147