INTERNATIONAL JOURNAL OF RESEARCH AND REVIEWS IN SOCIAL AND APPLIED SCIENCES

ISSN: 2931 – 7853 | www.caarnjournals.com | Volume 8, Issue 1. (AUGUST 2022)

DEADLOCK AVOIDANCE

BENJAMIN, UBOKOBONG EFFIONG & DR. MFREKE UMOH
Department of Computer Science,
Akwa Ibom State Polytechnic,
Ikot Osurua, Ikot Ekpene
benjaminubokobong52@gmail.com | mfreke_u@yahoo.com

Abstract

This seminar paper is focused on deadlock avoidance. Deadlock can be defined as a situation where a set of processes are blocked because each process is holding a resource and waiting for another resource acquired by some other process In deadlock avoidance, the request for any resource will be granted if the resulting state of the system doesn't cause deadlock in the system. In order to avoid deadlocks, the process must tell OS, the maximum number of resources a process can request to complete its execution. A deadlock prevention algorithm organizes resource usage by each process to ensure that at least one process is always able to get all the resources it needs. One such example of deadlock algorithm is Banker's algorithm.

Introduction

Deadlock avoidance is the process of avoiding deadlock. A deadlock is a state in which each member of a group waits for another member, including itself, to take action, such as sending a message or more commonly releasing a lock. Deadlocks are a common problem in multiprocessing systems, parallel computing, and distributed systems, where software and hardware locks are used to arbitrate shared resources and implement process synchronization. In an operating system, a deadlock occurs when a process or thread enters a waiting state because a requested system resource is held by another waiting process, which in turn is waiting for another resource held by another waiting process. If a process is unable to change its state indefinitely because the resources requested by it are being used by another waiting process, then the system is said to be in a deadlock. In a communications system, deadlocks occur mainly due to lost or corrupt signals resource contention. Two processes competing for two resources in opposite order (Ling, Chen and Chiang, 2006).

A. A single process goes through.
B. The later process has to wait.
C. A deadlock occurs when the first process locks the first resource at the same time as the second process locks the second resource.
D. The deadlock can be resolved by cancelling and restarting the first process.

ISSN: 2931 – 7853 | www.caarnjournals.com | Volume 8, Issue 1. (AUGUST 2022)

Deadlock can also be seen as a situation where a set of processes are blocked because each process is holding a resource and waiting for another resource acquired by some other process. Consider an example when two trains are coming toward each other on the same track and there is only one track, none of the trains can move once they are in front of each other. A similar situation occurs in operating systems when there are two or more processes that hold some resources and wait for resources held by other(s). For example, in the below diagram, Process 1 is holding Resource 1 and waiting for resource 2 which is acquired by process 2, and process 2 is waiting for resource 1.

Distributed deadlocks can occur in distributed systems when distributed transactions or concurrency control is being used. Distributed deadlocks can be detected either by constructing a global wait-for graph from local wait-for graphs at a deadlock detector or by a distributed algorithm like edge chasing (Tanenbaum, 1995). Phantom deadlocks are deadlocks that are falsely detected in a distributed system due to system internal delays but do not actually exist. For example, if a process releases a resource R1 and issues a request for R2, and the first message is lost or delayed, a coordinator (detector of deadlocks) could falsely conclude a deadlock (if the request for R2 while having R1 would cause a deadlock). Deadlock avoidance is therefore an important concept in operating systems. It is also known as deadlock prevention (Coffman, Elphick, and Shoshani, 1971).

Necessary Conditions for Deadlock to Occur

A deadlock situation on a resource can arise if and only if all of the following conditions occur simultaneously in a system (Padua, 2011):

1. *Mutual exclusion:* At least two resource must be held in a non-shareable mode. Otherwise, the processes would not be prevented from using the resource when necessary. Only one process can use the resource at any given instant of time.
2. *Hold and wait* or *resource holding:* a process is currently holding at least one resource and requesting additional resources which are being held by other processes.
3. *No preemption:* a resource can be released only voluntarily by the process holding it.
4. *Circular wait:* each process must be waiting for a resource which is being held by another process, which in turn is waiting for the first process to release the resource. In general, there is a set of waiting processes, $P = \{P_1, P_2, \ldots, P_N\}$, such that P_1 is waiting for a resource held by P_2, P_2 is waiting for a resource held by P_3 and so on until P_N is waiting for a resource held by P_1.

These four conditions are known as the *Coffman conditions* from their first description in a 1971 article by Edward G. Coffman, Jr. While these conditions are sufficient to produce a deadlock on single-instance resource systems, they only indicate the possibility of deadlock on systems having multiple instances of resources (Coffman, Elphick, and Shoshani, 1971).

ISSN: 2931 – 7853 | www.caarnjournals.com | Volume 8, Issue 1. (AUGUST 2022)

Deadlock Handling

Most current operating systems cannot prevent deadlocks. When a deadlock occurs, different operating systems respond to them in different non-standard manners (Padua, 2011). Most approaches work by preventing one of the four Coffman conditions from occurring, especially the fourth one. Major approaches are as follows:

- ***Ignoring deadlock*:** In this approach, it is assumed that a deadlock will never occur. This is also an application of the Ostrich algorithm. This approach was initially used by MINIX and UNIX. This is used when the time intervals between occurrences of deadlocks are large and the data loss incurred each time is tolerable. Ignoring deadlocks can be safely done if deadlocks are formally proven to never occur. An example is the RTIC framework.
- ***Detection*:** Under the deadlock detection, deadlocks are allowed to occur. Then the state of the system is examined to detect that a deadlock has occurred and subsequently it is corrected. An algorithm is employed that tracks resource allocation and process states, it rolls back and restarts one or more of the processes in order to remove the detected deadlock. Detecting a deadlock that has already occurred is easily possible since the resources that each process has locked and/or currently requested are known to the resource scheduler of the operating system. After a deadlock is detected, it can be corrected by using one of the following methods:

1. ***Process termination**:* one or more processes involved in the deadlock may be aborted. One could choose to abort all competing processes involved in the deadlock. This ensures that deadlock is resolved with certainty and speed. But the expense is high as partial computations will be lost. Or, one could choose to abort one process at a time until the deadlock is resolved. This approach has a high overhead because after each abort an algorithm must determine whether the system is still in deadlock. Several factors must be considered while choosing a candidate for termination, such as priority and age of the process (Padua, 2011).
2. ***Resource preemption**:* resources allocated to various processes may be successively preempted and allocated to other processes until the deadlock is broken.

***Deadlock Avoidance (Prevention)*:** Deadlock avoidance is the mostly used by several types of operating systems, but it is used mainly for end users. This concept is more comfortable for single user system because they use their system for simply browsing as well as other simple activities. Deadlock avoidance technique (method) helps to avoid deadlock to occur in the system.

Deadlock Avoidance Concept

To avoid deadlock, the operating system (OS) need to determine in advance safe and unsafe state for running processes. Each process provides OS with information about its requests and releases

ISSN: 2931 – 7853 | www.caarnjournals.com | Volume 8, Issue 1. (AUGUST 2022)

for resources Ri. OS decides whether deadlock will occur at run time e.g. batch jobs know a priori which resources they'll request and when (Schneider, 2009). A resource allocation state is defined by:

- Number of available resources.
- Number of allocated resources to each process
- Maximum demands by each process

When there is a request, the system determines whether allocating resources for the request leaves the system in a safe state that avoids deadlock

• if no, then wait for another process to release resources. Each process declares its maximum demands and it must be less than total resources in system. The Advantages of this are:

- Allow multiple types of resources
- Works for multiple instances of resources
- Adapts at run time to individual requests, avoids deadlock
- Individual requests don't need to be known a priori, except for maximum demands, which is not completely unrealistic.

A state is *safe* if there exists a *safe sequence* of processes <P1, ..., Pn> for the current resource allocation state. A sequence of processes is safe if for each Pi in the sequence, the resource requests that Pi can still make can be satisfied by:

- Currently available resources + all resources held by all previous processes in the sequence Pj, j<i.
- If resources needed by Pi are not available, Pi waits for all P j to release their resources.

Given that the system is in a certain state, we want to find at least one "way out of trouble" – i.e. find a sequence of processes that, even when they demand their maximum resources, won't deadlock the system. This is known as a worst-case analysis. It may be that during the normal execution of processes, none ever demands its maximum in a way that causes deadlock. To perform a more optimal (less than worst-case) analysis is more complex, and also requires a record of future accesses (Schneider, 2009). It is important to know that:

- A safe state provides a safe "escape" sequence.
- A deadlocked state is unsafe.
- An unsafe state is not necessarily deadlocked
- A system may make transition from a safe to an unsafe state if a request for resources is granted.
- Ideally, the OS check with each request for resources whether the system is still safe.

ISSN: 2931 - 7853 | www.caarnjournals.com | Volume 8, Issue 1. (AUGUST 2022)

Example 1:

- 12 instances of a resource
- At time t0, P0 holds 5, P1 holds 2, P2 holds 2
- Available = 3 free instance.

Table 1: Resource allocation table

Processes	Max Needs	Allocated
P0	10	5
P1	4	2
P2	9	2

Given the above, the following questions are important. Is the system in a safe state? Can a safe sequence be found? The answer is yes, the sequence <P1, P0, P2> is safe. This is because:

- P1 requests its maximum (currently has 2, so needs 2 more) and holds 4, then there is only 1 free resource.
- Then P1 then releases all of its held resources, so that there are 5 free.
- Next, suppose P0 requests its maximum (currently has 5, so needs 5 more) and holds 10, so that there are 0 free.
- Then P0 releases all of its held resources, so that there are 10 free
- Next P2 requests its max of 9, leaving 3 free and then releases them all – Thus the sequence <P1, P0, P2> is safe, and is able in the worst-case to request maximum resources for each process in the sequence, and release all such resources for the next process in the sequence (Bensalem, Fernandez, Havelund, and Mounier, 2006).

Example 2: At time t1, process P2 requests and is given 1 more instance of the resource, then:

ISSN: 2931 – 7853 | www.caarnjournals.com | Volume 8, Issue 1. (AUGUST 2022)

Processes	Max Needs	Allocated
P0	10	5
P1	4	2
P2	9	3

Table 2: Resource allocation table

From the table above, available = 2 free instances. The question again is, is the system in a safe state?

- P1 can request its maximum (currently holds 2, and needs 2 more) of 4, so that there are 0 free
- P1 releases all its held resources, so that available = 4 free
- Neither P0 nor P2 can request their maximum resources (P0 needs 5, P2 needs 6, and there are only 4 free)
- Both would have to wait, so there could be deadlock

Given this condition or state, the system is deemed unsafe. The mistake was:

- Granting P2 an extra resource at time t1.
- Forcing P2 to wait for P0 or P1 to release their resources would have avoided potential deadlock.

An important Policy to avoid safe state is that: before granting a request, at each step, perform a worst-case analysis to determine the following:

- Is there a safe sequence, a way out?

- If so, grant request.

- If not, delay requestor, and wait for more resources to be freed.

Deadlock avoidance works by preventing one of the four Coffman conditions from occurring.

- Removing the *mutual exclusion* condition means that no process will have exclusive access to a resource. This proves impossible for resources that cannot be spooled. But even with spooled resources, the deadlock could still occur. Algorithms that avoid mutual exclusion are called non-blocking synchronization algorithms (Coulouris, 2012).
- The *hold and wait* or *resource holding* conditions may be removed by requiring processes to request all the resources they will need before starting up (or before embarking upon a particular set of operations). This advance knowledge is frequently difficult to satisfy and, in any case, is an inefficient use of resources. Another way is to require processes to request resources only when it has none; First, they must release all their currently held resources before requesting all the resources they will need from

ISSN: 2931 – 7853 | www.caarnjournals.com | Volume 8, Issue 1. (AUGUST 2022)

scratch. This too is often impractical. It is so because resources may be allocated and remain unused for long periods. Also, a process requiring a popular resource may have to wait indefinitely, as such a resource may always be allocated to some process, resulting in resource starvation. (These algorithms, such as serializing tokens, are known as the *all-or-none algorithms*.)

- The *no preemption* condition may also be difficult or impossible to avoid as a process has to be able to have a resource for a certain amount of time, or the processing outcome may be inconsistent or thrashing may occur. However, the inability to enforce preemption may interfere with a *priority* algorithm. Preemption of a "locked out" resource generally implies a rollback, and is to be avoided since it is very costly in overhead. Algorithms that allow preemption include lock-free and wait-free algorithms and optimistic concurrency control. If a process holding some resources and requests for another resource(s) that cannot be immediately allocated to it, the condition may be removed by releasing all the currently being held resources of that process (Bensalem, *et al*, 2006).
- The final condition is the *circular wait* condition. Approaches that avoid circular waits include disabling interrupts during critical sections and using a hierarchy to determine a partial ordering of resources. If no obvious hierarchy exists, even the memory address of resources has been used to determine ordering and resources are requested in the increasing order of the enumeration (Coulouris, 2012).

Banker's Algorithm for Deadlock Avoidance

Banker's Algorithm is used for resource allocation and deadlock avoidance that determine safety by simulating the allocation for predetermined maximum possible amounts of all resources, then makes an "s-state" check to test for possible activities, before deciding whether allocation should be allowed to continue. Banker's Algorithm is named because it is implemented in the banking sector, and main objective of using this algorithm is to check that loan can be sanctioned or not to clients. Bankers's Algorithm test all the request made by processes for resources, it checks for the safe state, if after granting request system remains in the safe state it allows the request and if there is no safe state it doesn't allow the request made by the process (Stuart, 2008).

The Banker's Algorithm is as follows:

- When there is a request, the system determines whether allocating resources for the request leaves the system in a safe state that avoids deadlock
- If no, then wait for another process to release resources
- Each process declares its maximum demands and it must be less than total resources in system.

ISSN: 2931 – 7853 | www.caarnjournals.com | Volume 8, Issue 1. (AUGUST 2022)

The advantages of bankers algorithm are:

- To allow multiple types of resources.

- Works for multiple instances of resources.
- Adapts at run time to individual requests, avoids deadlock.
- Individual requests don't need to be known a priori, except for maximum demands, which is not completely unrealistic.

The inputs to Banker's Algorithm are:

1. Max need of resources by each process.
2. Currently, allocated resources by each process.
3. Max free available resources in the system.

The request will only be granted under the below condition:

1. If the request made by the process is less than equal to max need to that process.
2. If the request made by the process is less than equal to the freely available resource in the system.

Example 3:

A process in operating system uses resources in the following way.
1) Requests a resource

2) Use the resource

3) Releases the resource

Conclusion

Deadlock avoidance is the technique of preventing deadlock from occurring. Deadlock can also be seen as a situation where a set of processes are blocked because each process is holding a resource and waiting for another resource acquired by some other process. To prevent this, deadlock avoidance algorithm such as Bankers' algorithm is used. The algorithm is used to determine safe and unsafe states for running processes in advance so as to avoid deadlock. With a deadlock avoidance algorithm in place, processes will be allocated available resources so long as it is safe to do so.

Recommendations

The following recommendations are offered based on the study conducted:

- Operating systems should be designed to implement deadlock avoidance algorithms.

ISSN: 2931 – 7853 | www.caarnjournals.com | Volume 8, Issue 1. (AUGUST 2022)

- Existing deadlock avoidance algorithms should be improved upon to be more optimal.
- More research on deadlock avoidance should be encouraged.

REFERENCES

Bensalem, S., Fernandez, J., Havelund, K. & Mounier, L. (2006). Confirmation of deadlock potentials detected by runtime analysis. *Proceedings of the 2006 Workshop on Parallel and Distributed Systems: Testing and Debugging. ACM*. pp. 41–50.

Coffman, E., Elphick, M. and Shoshani, A. (1971). System deadlocks. *ACM Computing Surveys*. **3** (2): 67–78.

Coulouris, G. (2012). *Distributed systems concepts and design*. Pearson, NY.

Ling, Y., Chen, S. and Chiang, J. (2006). On optimal deadlock detection scheduling. *IEEE Transactions on Computers*. **55** (9): 1178–1187.

Padua, D. (2011). *Deadlock management techniques*. Encyclopedia of Parallel Computing. Springer. p. 524.

Schneider, G. (2009). *Invitation to computer science*. Cengage Learning. p. 271.

Shibu, K. (2009). *intro to embedded systems* (1st ed.). Tata McGraw-Hill Education. p. 446.

Silberschatz, A. (2006). *Operating system principles* (7 ed.). Wiley-India.

Stuart, B. (2008). *Principles of operating systems* (1st ed.). Cengage Learning. p. 446.

Tanenbaum, A. (1995). *Distributed operating systems* (1st ed.). Pearson Education. p. 117.

https://digitalthinkerhelp.com/deadlock-avoidance-algorithms-in-os-operating-system-must-be-known/