International Journal of Engineering and Applied Computer Science (IJEACS)
Volume: 02, Issue: 02, February 2017

Improved Deadlock Prevention Algorithms in Distributed Systems

Mahboobeh Abdoos
Electrical and Computer Engineering Department
Qom Islamic Azad University, Qom, Iran.

***Abstract*—Distributed systems deadlock is similar to single-processor system deadlock, but is worse. It is harder to avoid, prevent or detect and is harder to cure, when it is tracked down because all the relevant information is scattered over many machines. In some systems, such as distributed database systems, it can be extremely serious, so it is important to understand how it differs from ordinary deadlock and what can be done about it. Two important deadlock prevention algorithms in distributed systems are wait-die and wound-wait. Their problem is that they just attend to the time stamp of processes, but not priority of them. In a real operating system, attending to priority of processes is very important. The proposed improved algorithms are attending to both priority and time stamp of processes.**

Keywords —Deadlock Prevention; Distributed Systems; Algorithm; Process.

I. INTRODUCTION

During the last decade computing systems have undergone a rapid development, which has a great impact on distributed operating systems. While commercial systems are gradually maturing, new challenges are imposed by the world-wide interconnection of computer systems. This creates an ever growing need for large-scale enterprise distributed solutions. In future, distributed operating systems will have to support hundreds or even thousands of sites and millions of clients and, therefore, will face tremendous scalability challenges with regard to performance, availability and administration. One of the challenges that we must solve it in this area is deadlock problem. Deadlock also is one of the most serious problems in multitasking concurrent programming systems.

The rest of the paper organized as follow. In Section 2 it is briefly described the deadlock and its scope in distributed operating systems, there is an overview of distributed systems deadlock prevention algorithms. Section 3 presents improved deadlock prevention algorithms. Finally we summarize at section 4.

II. DISTRIBUTED SYSTEMS DEADLOCK

A deadlock is an undesirable situation where members of a set of processes that hold resources are locked indefinitely from access to resources are blocked indefinitely from access to resources held by other members within the set. No member of the set can release its own resources before completing its tasks. Therefore the deadlock will last forever, unless a deadlock resolution procedure is performed. If the involved processes in a deadlock are spread out in a network of computers or in a distributed computer system. The situation is known as distributed deadlock.

An approach for handling the deadlocks is to prevent them from occurring in a system. The basic idea behind a deadlock prevention algorithm is that it ensures at least one of the conditions necessary for deadlock to occur can not hold. A common technique is to prevent processes from waiting for each other in a circular manner. [1]

As already mentioned, a simple approach to deadlock prevention is to refuse the request of process (by aborting the process) for a resource which is currently held by another process. This approach requires a transaction mechanism which ensures that the result of a process is either complete or has no effect, even in the presence of failures (i.e. Interrupted by accident or aborted deliberately by a system). For the rest of the paper we use transactions instead of processes as units of execution.

This condition is called deadlock. Deadlock has four conditions:

1. Mutual exclusion: Each resource can only be assigned to exactly one resource.
2. Hold and wait: Processes can hold resources and request more.
3. Non preemption: Resources can not be forcibly removed from a process.
4. Circular wait: There must be a circular chain of processes, each waiting for a resource held by the next member of the chain.

A. Deadlock Problem Solving Approaches

There are four techniques commonly employed to deal with deadlocks:

1. Ignore the problem
2. Deadlock detection
3. Deadlock prevention
4. Deadlock avoidance

Ignoring deadlocks is the easiest scheme to implement. Deadlock detection attempts to locate and resolve deadlocks. Deadlock avoidance describes techniques that attempt to determine if a deadlock will occur at the time a resource is requested and react to the request in a manner that avoids the deadlock. Deadlock prevention is the structuring of a system in such a manner that one of the four necessary conditions for deadlock cannot occur [8]. Each solution category is suited to a specific type of environment and has advantages and disadvantages. In this paper, we focus on deadlock prevention which is the most commonly implemented deadlock solution. Deadlock prevention in distributed systems includes two algorithms named wait-die and wound-wait.

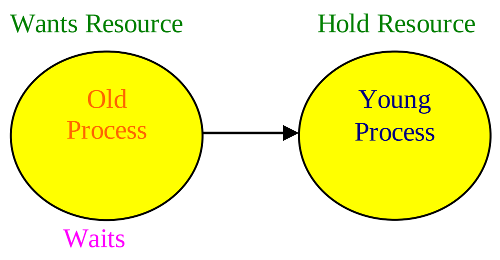
B. Wait-Die Algorithm

Transactions are ordered by timestamps. A transaction is stamped at its creation. Ordering between different transactions is identical on all execution nodes. Let Ti and Tj be two transactions marked with time stamps TS(Ti) and TS(Tj).

In this algorithm if the old process is waiting for the young process it can wait, else if the young process is waiting for the old process, the young process will die and restart again with the same time stamp. This algorithm just attends to the time stamp of processes and not the priority of them. Fig.1 shows the wait-die algorithm [3].

wait-die:

Ti ----- > Tj
if TS(Ti)<TS(Tj)
(Ti is older than Tj) then Ti is allowed to wait otherwise (Ti younger than Tj) Ti is aborted and is restarted later with the same timestamp



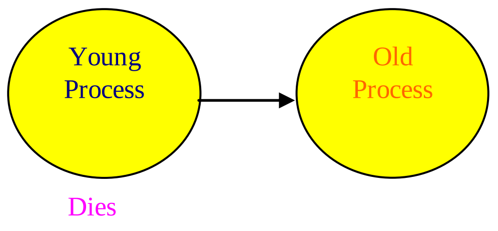


Figure 1. Wait-Die Algorithm

C. Wound-Wait Algorithm

Here like the before algorithm, each process has an individual time stamp. And two processes have not the same time stamp. In this algorithm, if the old process is waiting for the young process, it can wound the young process [7] and get its resources. If the young process is waiting for the old process, it can wait. Fig.2 shows the wound-wait algorithm.

wound-wait:

Ti ----- > Tj
if TS(Ti)<TS(Tj)
(Ti older than Tj) then Ti wounds Tj and get its resources.
Otherwise (Ti younger than Tj)Ti is allowed to wait

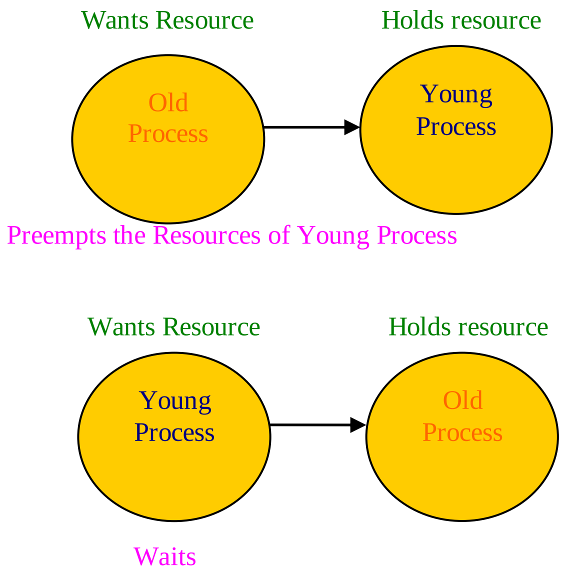


Figure. 2 Wound-Wait Algorithm

III. Proposed Algorithms

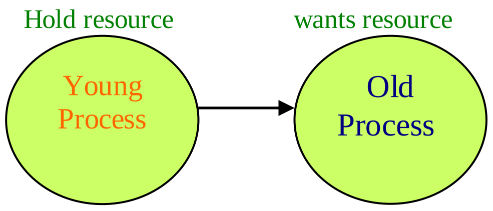
Proposed-wait-die:
Ti ----- > Tj
If TS(Ti)<TS(Tj) (Ti is older than Tj) then Ti is allowed to wait .
If TS(Ti)>TS(Tj) (Ti younger than Tj) and (priority(Ti)<priority(Tj)) then Ti is aborted and is restarted later with the same timestamp.
if TS(Ti)>TS(Tj) (Ti younger than Tj) and (priority(Ti)>priority(Tj)) then it is allowed to wait.

Proposed-wound-wait:
if Ts(Ti)<TS(Tj)
(Ti older than Tj) and (priority(Ti)>priority(Tj)) then Ti wounds Tj and get it's resources.
if (Ti older than Tj) and (priority(Ti)<priority(Tj)) then the old process continues to wait.
If TS(Ti)>TS(Tj) (Ti younger than Tj) Ti is allowed to wait

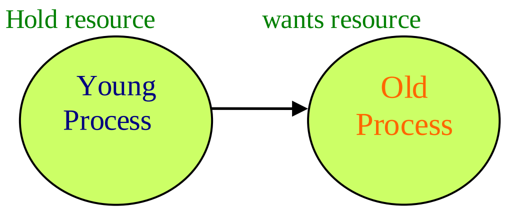
One important problem of wait-die and wound-wait algorithms is that they do not attend to the priority of processes. Figures 3, 4 show the proposed algorithms.

In the proposed wait-die algorithm, if the old process is waiting for the young process it can wait and if the young process is waiting for the old process and the priority of young process is lower than the priority of old process then the young process will die and restart again with the same time stamp. If the young process is waiting for the old process and the priority of the young process is higher than the priority of the old process, then the young process continues to wait.

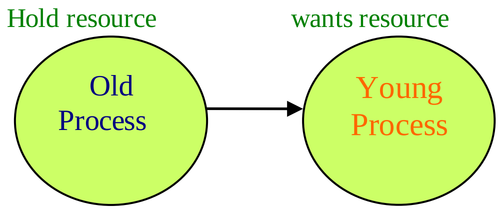
In the proposed wound-wait algorithm if the old process is waiting for the young process then if the priority of old process is more than the priority of young process then the old process kills the young process and preempts its resources. If the old process is waiting for the young process and the priority of young process is higher than the priority of old process then the old process is allowed to wait. These proposed algorithms attend to both priority and time stamp of processes.



(a) The young process dies. (If the priority of young process is lower than the priority of old process)

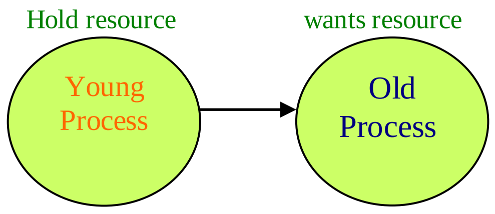


(b) The young process can wait. (If the priority of young process is higher than the priority of old process)

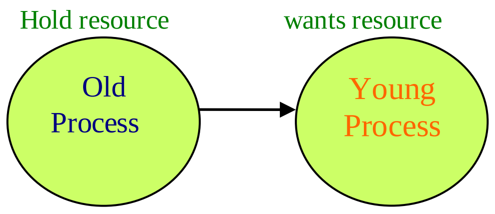


(c) The old process waits.

Figure 3. The proposed wait-die algorithm

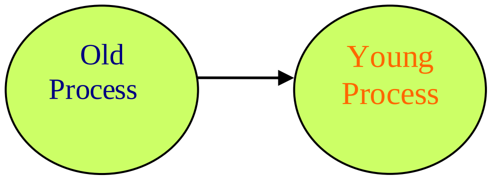


(a) The young process waits.



(b) The old process preempts the resources of the young process. (If the priority of old process is higher than the priority of young process)

Hold resource wants resource



(c) The old process waits. (If the priority of old process is lower than the priority of young process)

Figure 4. The proposed wound wait algorithm

IV. CONCLUSION

In this paper, it is considered to the deadlock problem in distributed systems and reviewed the approaches like wait-die and wound wait algorithms. One important problem of wait-die and wound-wait algorithms is that they don't attend to the priority of processes. In the proposed wait-die algorithm, if the old process is waiting for the young process it can wait and if the young process is waiting for the old process and the priority of young process is lower than the priority of old process then the young process will die and restarts again with the same time stamp. If the young process is waiting for the old process and the priority of the young process is higher than the priority of the old process, then the young process continues to wait. In the proposed wound-wait algorithm if the old process is waiting for the young process then if the priority of old process is more than the priority of young process then the old process kills the young process and preempts its resources. If the old process is waiting for the young process and the priority of young process is higher than the priority of old process then the old process is allowed to wait. These proposed algorithms attend to both priority and time stamp of processes.

REFERENCES

[1] Gregory R. Andrews, Gary M. Levin," On-the-fly Deadlock Prevention", ACM Computing surveys, 1982.

[2] Wilson C.H," Using Ordered and Atomic Multicast for Distributed Deadlock Prevention".

[3] N. De Palma, P. Laumay, L. Bellissard, "Ensuring Dynamic Reconfiguration Consistency".

[4] Anna Hac, Xiaowei Jin, Jo-Han Soo, "A Performance Comparison of Deadlock Prevention and Detection Algorithms in a Distributed File System", In Proc Of 8th Annual Inter Phoenix Conference On Computers and Communications, pp. 473-477, 1989.

[5] L. Lamport, Time, "clocks and the ordering of the events in a distributed system", Common of ACM, 21(7): 558-565, 1978.

[6] Richard C. Holt," Comments on Prevention of System Deadlocks", Common of the ACM, volume 14, number 1, 1971.

[7] C.J. Date, "In Introduction to Database systems", volume II, 1983.

[8] Mahdi Samadi, "Survey of Deadlock Detection In Distributed Operating System".

AUTHOR PROFILE

Mahboobeh Abdoos received the B.S and M.S degrees in computer engineering from Azad University, Ghazvin, Iran, in 2002 and 2007 respectively. She is now the Ph.D. research student of Islamic Azad university, Qom, Iran. She has taught at Islamic Azad and Payam Nour Universities from 2005 til now. She has been the referee of some conferences. Her current research interest includes position based routing protocols in mobile ad hoc networks, QOS and security based routing protocols in mobile ad hoc networks, cloud computing and data base.