다중처리 시스템의 병렬성 증대를 위한 사이클의 비 지연 발견 기법

(A Zero-latency Cycle Detection Scheme for Enhanced Parallelism in Multiprocessing Systems)

김 주 균 †

(Ju Gyun Kim)

요 약 본 논문에서는 즉시 할당 상태와 함께 단일 자원, 단일 요청의 가정 하에서 다중처리 시스템에서 사이클을 발생 즉시 발견함으로써 지연 없는 교착상태의 발견 방법을 소개한다. 기존의 방법과는 달리 제시된 방법은 n과 m으로 프로세스와 자원의 수를 나타낼 때 사이클의 발견에 O(1), 대기나 자원의 반납 시에 O(n+m)의 시간을 요한다. 따라서 n과 m의 크기에 상관없이 교착상태를 발생 즉시 알 수 있으며, 이 점이 다중처리 시스템의 특성과 잘 조화될 수 있음을 보였다. 교착상태와 연관된 응용환경에서 이러한 발견의 예측성과 비 지연성은 매우 유용할 것이다.

키워드 : 교착상태, 비 지연, 사이클 발견, 병렬성

Abstract This paper presents a non-blocking deadlock detection scheme with immediate cycle detection in multiprocessing systems. We assume an expedient state and a special case where each type of resource has one unit and each request is limited to one resource unit at a time. Unlike the previous deadlock detection schemes, this new method takes O(1) time for detecting a cycle and O(n+m) time for blocking or handling resource release where n and m are the number of processes and that of resources in the system. The deadlock detection latency is thus minimized and is constant regardless of n and m. However, in a multiprocessing system, the operating system can handle the blocking or release on-the-fly running on a separate processor, thus not interfering with user process execution. To some applications where deadlock is concerned, a predictable and zero-latency deadlock detection scheme could be very useful.

Key words : Deadlock, Zero-latency, Cycle detection, Parallelism

1. 서 론

병렬처리를 포함한 다중처리 시스템에 관한 연구는 지속적으로 이루어져 왔으나 이러한 시스템에서 사용 가능한 교착상태와 관련된 연구는, 자원의 수에 비해 상대적으로 늘어난 프로세스 개수에 비례하여 발생 가능성 또한 증가됨에도 불구하고, 크게 주목 받지 못하였다. 따라서 본 연구는 다중처리 시스템 하에서 사이클(Cycle)을 발생 즉시 발견하여 교착상태의 판단을 즉각적으로 할 수 있는 방법을 소개한다.

프로세스가 사용 가능한 상태의 자원(Resource)을 요구할 때 지체 없이 할당(*Allocation*)해 주는 상태를 즉시 할당 상태(Expedient state)라 하며 거의 대부분의 시스템들은 이런 상태를 가정한다[1]. 이러한 가정 하에서 프로세스가 한번에 하나의 자원을 요구할 수 있도록 하면 즉, 필요한 자원을 순서대로 요청하도록 한다면 노트(Knot)의 발견이 교착상태 발견의 필요충분조건이 되며[1-4] 이 경우의 O(1) 노트 발견 방법은 저자에 의해 이미 소개되었다[5-7].

위의 가정에 더하여 하나의 자원 유형(Type)에 하나의 자원(Single unit)만을 가지도록 조건을 강화하면 사이클의 발견이 교착상태 발견의 필요충분조건이 된다. 이 경우 노트의 발견에 비해 더 적은 오버헤드(Overhead) 즉, 교착상태의 처리를 위해 소요되는 시간의 단축과 필요한 자료구조를 위해 요구되는 메모리의 양적 감소와 같은 이점을 가지는 교착상태 발견 방법을 시도할 수 있다.

· 본 연구는 숙명여자대학교 2004년도 교비연구비 지원에 의해 수행되었음

† 정 회 원 : 숙명여대 정보과학부 교수

jgkim@sookmyung.ac.kr

†† 논문접수 : 2004년 5월 4일

심사완료 : 2004년 11월 2일

사이클의 발견을 위해 기존의 연구들이 취하는 방법은, 연속적 발견(Continuous detection)법을 따라도, 프로세스가 비가용(Unavailable) 자원을 요청한 후 대기(Block)될 때 교착상태의 발견을 위한 알고리즘이 구동된다. 이 경우 교착상태의 발생 유무에 대한 판단은 시스템의 상황을 자원 할당 그래프(Resource allocation graph: RAG)로 나타냈을 때 탐색해야할 에지(Edge)의 수만큼 -n과 m으로 프로세스와 자원의 수를 나타낼 경우 최대 O(n+m) - 이 후에 가능하다. 즉, 이 시간동안 해당 프로세스는 어떤 처리도 받지 못하게 되며 이러한 문제는 다중처리 시스템이라 할지라도 달라질 수 없다.

다중처리가 가능한 환경에서 하나의 프로세서(Processor)를 교착상태 발견용으로 지정했을 때를 가정해 보자. 발견용 프로세서는 프로세스들이 대기될 때 구동될 것이며, 대기된 프로세스는 발견의 결과에 따라 처리가 달라질 것이다. 즉, 단순한 대기 상태라면 다른 프로세스로의 교체(Switching)로 해결되지만 교착상태라면 그에 따른 복구 절차가 따르게 된다. 다시 말해 판단의 결과에 따라 처리가 달라질 수 있는 상황에서 대기상태를 유발한 프로세스와 발견을 위해 구동되는 프로세스는 이 후 결과가 나올 때까지는 동시에 진행될 수 없게 됨으로써 이 기간 동안에는 다중처리의 병렬성을 기대할 수 없게 된다.

좀 더 구체적인 예로서 다수개의 프로세스가 서로 정보를 주고받으며 동시에 실행중인 상황을 가정해 보자. 이 중 하나의 프로세스가 대기 상태가 될 경우 교착상태를 발생 즉시 알 수 있다면 이 시점에서 필요한 모든 조치가 바로 실행되어 질 수 있을 것이다. 반대로 교착상태의 판단에 O(n+m) 시간이 요구된다면 이 시간동안에 다른 프로세스들은 계속해서 진행될 것이며 판단의 결과 교착상태임이 밝혀지고 복구를 위해 대기 상태의 프로세스가 처음부터 다시 시작해야할 상황이 될 경우 이미 이 프로세스의 대기 상태 이전에 정보를 주고받았던 다른 프로세스들 역시 도미노 효과(Domino effect)에 의해 최악의 경우 처음으로 되돌아가야 하거나 최소한 O(n+m)시간 동안의 실행 부분은 취소될 것이다. 다시 말해, 대기 상태가 될 경우 교착상태를 발생 즉시 알 수 있었다면 취소되는 작업량의 손실은 겪지 않아도 될 것이며 이 기간에 가동되었던 해당 프로세서들은 다른 유용한 작업을 위해 동원될 수 있었을 것이다.

결론적으로, 기존의 방법들은 그 속성상 교착상태의 유무를 판단하는데 O(n+m)의 시간을 요구하게 되며 이 점이 다중처리 시스템이 제공할 수 있는 병렬처리의 장점을 감소시키는 요인이 된다. 응용 분야에 따라서는, 예를 들어 실시간(Realtime) 시스템의 경우에도, 특정 프로세스를 대기상태로 만드는 요청이 단순히 대기상태인지 교착상태인지를 즉시 알 수 있다는 점이 매우 중요하게 다뤄질 것이다.

사이클의 발견을 위해 자원 할당 그래프가 유지되어야하고 이 그래프는 프로세스의 다양한 행동 이를테면, 자원의 요청, 할당, 대기, 반납에 따라 지속적으로 수정이 필요하게 되는데 이 과정에서 O(n+m)의 시간이 요구된다. 결국 교착상태의 처리를 위해 O(n+m)의 시간이 들 수밖에 없는 상황에서 이 시간을 어디에 활용하는 것이 다중처리 시스템에 유리하고 또, 그럴 가능성은 있는지가 관건이다. 만약 교착상태의 판단을 O(1) 시간으로 하고 단순한 대기 상태나 자원의 반납(Release)과 같은 일의 처리에 O(n+m)의 시간이 필요하도록 할 수 있다면 지적한 기존 방법들의 문제점을 극복할 수 있을 것이라는 판단 하에 본 논문에서는 몇 가지의 가정과 함께 자료구조 및 알고리즘을 소개한다. 2장에서 관련연구를, 3장은 시스템의 상태를 설명하고 4장에서 알고리즘과 함께 각 부분에서 소요되는 시간에 관한 분석을 보이며 끝으로 5장에서 결론을 언급하였다.

2. 관련 연구

Shoshani와 Holt[4,8]의 연구는 교착상태의 처리와 관련하여 기초가 되는 이론을 제시했다는 점에서 주목할 부분이다. 특히 Holt는 즉시 할당 상태의 가정 하에서 단일 요청의 경우는 노트가, 단일 요청(Single request) 및 단일 자원(Single unit)의 경우에는 사이클이 교착상태의 필요충분조건이 됨을 보였으며 각각 O(nm)과 O(n+m)이 소요됨을 보였다. 동시에 다수개의 자원을 요청하는 다중 요청의 경우는 자원중 하나라도 만족되지 않을 경우 대기상태가 된다는 점에서 순서대로 하나씩 요청하는 것으로 적용하여도 문제가 없으며, 단일 자원의 가정은 각각의 자원을 동일한 유형이라 하더라도 독립적인 개별자원으로 정의하고 적용(Implement)하면 되므로 제한적이기는 하나 응용 분야에 따라 필요한 가정이다. 교착상태의 발견을 크게 사이클과 노트의 발견으로 나누는데, 대부분의 연구에서 언급하고 있는 사이클의 발견은 단일 자원이 가정되었을 때 가능하다는 점이 이를 반증한다.

Leibfried는 시스템의 상태를 인접행렬(Adjacency matrix)로 나타내고 이들의 반복적인 곱을 계산함으로써 교착상태를 발견하는 방법을 제시하였는데 프로세스의 개수에 비례하는 곱을 수행해야하므로 효과적이지 못하다[9].

프로세스와 자원간의 요청 및 할당상태를 그래프로 나타내면 그 속성상 양분 그래프(Bipartite graph)가 되며[4] 기존 연구들은 이 그래프에서 사이클의 유무를 판단하기위해 대기상태가 되는 프로세스로부터 연결된 에

지를 활동 프로세스(Active process) - 대기상태가 아닌 즉, 실행 가능한 프로세스를 말하며 이 후부터 싱크(Sink)라고 부른다 - 가 나올 때까지 계속 탐색해 나가는 방법을 취해왔다[1,2,5,6]. 이런 방법에서는 최악의 경우 수행시간은 당연히 양분그래프의 최대 에지 개수에 비례하는 O(n+m)의 시간 복잡도를 가질 수밖에 없다[1,3,10].

결과적으로 기존 연구의 단점은 몇 가지로 요약될 수 있는데 먼저, 교착상태의 발견에 소요되는 시간이 프로세스와 자원의 수에 의존(Dependant)함으로써 발견에 소요되는 시간이 예측 가능하지 않다는 점이다. 다음으로, 이런 연구들이 다중처리 시스템을 염두에 두고 개발되지 않았다는 것이 결국 시스템이 제공할 수 있는 병렬성을 취할 가능성을 낮추게 되는 요인이 된다. 더욱이 실시간 시스템과 같이 교착상태의 즉각적인 발견이 요구되는 경우에도 사용하기 힘들다.

[8]을 근거로 소수(Prime number)의 특성을 살려 교착상태의 즉시 발견을 시도한 [11]은 [12]와 마찬가지로 사용된 자료구조의 한계와 이것에 종속된 알고리즘의 특성 때문에 노트의 즉시 발견을 위한 알고리즘[6]으로의 자연스러운 확장이 불가능하다.

언급한 바와 같이 본 논문에서 소개되는 방법은 교착상태의 발견에 O(1)의 시간이, 단순 대기나 자원의 반납을 위한 자료구조의 수정에 O(n+m)의 시간이 소요되도록 함으로써 교착상태의 발견이 n과 m의 크기에 상관없이 예측 가능한 상수 시간 복잡도(Constant time complexity)를 가지도록 하였다. 첨언하면, 교착상태의 발견에 관한 한 기존의 연구와는 요구되는 시간의 측면에서 저자에 의해 처음으로 그리고 반대의 각도에서 접근하였으므로 동일한 관점에서 출발하는 연구를 찾고 비교할 수 없다는 것이 아쉬운 점이다.

소개된 알고리즘에서 O(n+m)의 시간이 요구되는 부분은 다중처리 시스템에서 여분의 프로세서를 교착상태 탐지용으로 활용하여 사용자 프로세스의 실행과 병렬적으로 실행되도록 함으로써 각자의 진행을 방해하지 않도록 하였으며, 자원을 반납할 경우 역시 반납한 프로세스의 계속적인 실행과 반납 때문에 필요한 자료구조의 수정 작업이 독립적이고 병렬적으로 수행 가능함으로써 병렬성의 증대와 함께 자원의 활용도 또한 높일 수 있게 된다.

3. 시스템 모델

언급되었듯이 사이클의 발견이 교착상태 발견의 필요충분조건이 되기 위해서는 몇 가지의 가정이 필요하다. 먼저, 즉시 할당 상태와 단일 요청 즉, 한번에 하나씩 자원을 요청하도록 해야 한다. 단일 요청은 동시에 다수개의 요청이 있을 경우 이를 순차화(Serialize) 하여 차례로 요청하도록 해줌으로써 충분히 수용 가능한 가정이다. 다음으로 단일 자원의 가정이 필요한데 이 제약은 노트의 발견에 비해 보다 간단한 자료구조와 알고리즘으로 사이클을 발견할 수 있다는 장점으로 상쇄가 가능하며 이런 경우의 예는 [1]에서 발견할 수 있다.

이상의 가정 하에서 시스템의 상태 - 자원의 할당과 대기를 나타내는 상태 - 를 RAG로 표현하면 다수개의 트리(Tree)로 구성됨을 알 수 있다. RAG내의 트리 하나를 관찰해 보면 각 노드(Node)는 많아야 하나의 나가는(Outgoing) 에지를 가지게 되는데 프로세스 노드의 경우에는 자신을 대기상태로 만든 자원노드와의 사이에, 자원 노드의 경우에는 자신이 할당된 프로세스 노드와의 사이에 에지가 존재할 것이다. 각 노드로 들어오는(Incoming) 에지는 프로세스의 경우 이미 할당된 자원노드로부터이며, 자원일 경우 이 자원 때문에 대기상태가 된 프로세스 노드들로부터가 될 것이다.

그림 1은 RAG를 구성하는 하나의 트리를 예시한 것으로 두 개의 자원 R1과 R2가 프로세스 P1에 할당되었음을 보여주고 있으며, P2와 P3는 R1에 의해 대기상태임을 알 수 있다. R5와 R6 또한 P3에 할당되었으나 아직은 사이클이 존재하지 않음을 알 수 있다. 이 트리에서 P1은 루트(Root)노드이며 유일한 싱크이다. 다시 말해 P1만이 나가는 에지가 없으며 다른 프로세스들은 전부 대기상태임을 알 수 있다. 결과적으로 각 트리의 루트노드만이 싱크이며 이들의 자원에 대한 요청과 반납 요구가 트리의 모양을 변화시킬 수 있을 것이다.

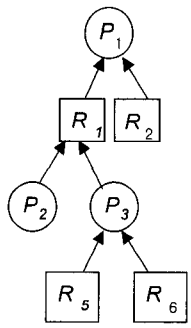


그림 1 RAG를 형성하는 트리의 예. 그림에서 각 자원들은 자신이 속한 트리의 루트 값 인 1을 가진다.

RAG내에 존재하는 다수개의 트리를 구분하기 위해서 각 트리는 자신의 루트노드 값을 가지게 하고 특정 트리에 속한 자원들은 자신이 속한 트리의 값을 가지도록 하는데 이 부분이 본 논문의 핵심이며 이 값의 활용은 다음 장의 *History* 행렬과 알고리즘에서 상세하게 다룰 것이다.

루트노드의 행동에 따라 교착상태 즉, 사이클의 발생이 있게 되는데 바로 자신이 루트인 트리의 값을 가지고 있는 자원을 요청한 경우이다. 그 외의 경우라면 가용자원을 요청했거나 비 가용자원의 경우라도 단순한 대기상태로 그칠 것이다.

결국 사이클은 싱크가 비가용 자원을 요청할 때 이 자원이 가지고 있는 소속 트리 값을 확인하여 싱크 자신의 값과 동일하면 발견되는 것이며, 이러한 확인 작업에 O(1)의 시간만 있으면 충분하다. 어떤 자원이 소속 값을 가지고 있지 않다는 것은 할당이 가능한 상태임을 말하며, 요청한 싱크와 값이 틀리다는 것은 다른 트리에 소속되어 있고 따라서 사이클이 형성되지 않으므로 요청한 싱크는, 교착상태가 아닌, 대기상태가 될 것이다. 물론 이 경우 다음 차례의 O(1) 사이클 발견이 보장되기 위해 해당 자료구조의 조정이 필요할 것이며 이는 다음 장의 알고리즘 설명에서 다룬다.

4. 알고리즘과 분석

자원에 대한 요청은 싱크만이 가능하며 각 트리의 루트노드들이 싱크들이 됨은 이미 밝혔다. 그림 2는 싱크의 자원에 대한 요청 형태를 나타낸 것으로 자원에 대한 어떠한 요청도 이 세 가지 범주를 벗어나지 않는다. 그림 2의 (a)는 가용자원 R_a에 대한 요청으로 즉시 할당 상태를 가정하였으므로 바로 할당해 주면 될 것이다. 물론 이 경우 R_a는 자신의 소속 값을 소속된 트리의 루트 값인 i를 가지도록 해야 한다. (b)와 같은 형태의 요청은 R_b가 다른 트리에 소속되어 있는 자원으로 P_i를 대기상태로 만들게 될 것이다. 이 경우 사이클은 형성되지 않으므로 교착상태가 아님은 자명하며 트리의 조정이 필요하게 된다. 즉, i를 소속 값으로 가지고 있던 모든 자원들은 자신의 소속 값을 j로 수정해야 함으로써 결과적으로 두 트리가 P_j가 루트인 하나의 트리로 합쳐지게 된다. (c)에서는 자신이 루트인 트리의 자원을 요구하게 됨으로써 사이클이 형성되어 교착상태가 됨을 알 수 있다.

싱크의 자원 반납은 교착상태와는 무관한 행동으로

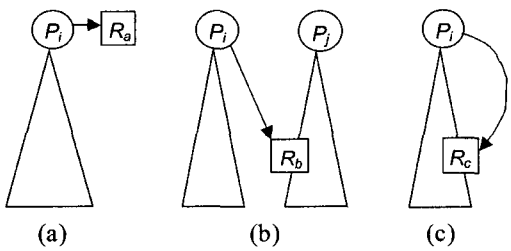


그림 2 프로세스의 자원에 대한 세 가지 요청 형태

역시 세 종류가 있다. 그림 3에서 P_i는 자신이 사용하던 자원을 반납하게 되는데 (a)의 경우 R_j에 대기되어 있는 프로세스가 없는 상황이므로 단순히 할당 에지를 없애는 작업 - R_j의 소속값인 i를 삭제하는 작업 - 이 필요하게 된다. (b)에서 R_j의 반납은 대기상태였던 P_k를 R_j를 할당해줌으로써 깨우게(Wakeup)되고 P_k는 새로운 트리의 루트가 됨과 동시에 자신의 밑에 있던 자원들(descendents)과 R_j의 소속 값은 k로 바뀌어야 한다. (c)에서는 다수개의 프로세스가 대기 중인 R_j를 반납하게 되고 이 경우에는 R_j를 할당받게 될 프로세스가 선택되어질 것이다. P_k가 선택된다고 가정하면 P_k를 루트로 하는 새로운 트리가 원래의 트리로부터 분리되어 만들어지게 되고 R_j와 그 이하 - R_j로의 경로(Path)가 있는 - 모든 노드들은 자신들의 소속 값을 k로 바꾸어야 한다. 물론 P_i가 루트인 본래의 트리에서 왼쪽 부트리(*Left subtree*)는 달라질 내용이 없다.

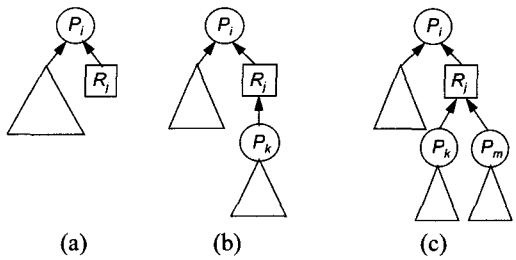


그림 3 프로세스의 자원에 대한 세 가지 반납 형태

(b)와 (c)의 작업은 최악의 경우 O(n+m)의 시간이 소요되며 이는 향 후 교착상태의 발견을 O(1) 시간에 가능케 하기 위해 필요하다. 그러나 자원의 반납은 교착상태와도 무관하고, 반납한 프로세스의 계속적인 실행을 방해하는 일도 아니므로 여분의 프로세서를 동원하여 병렬성을 높이는데 아무런 문제가 없을 것이다. 다음은 위에서 언급한 내용을 알고리즘으로 옮기기 위해 동원되는 자료구조를 소개한다.

① *History*(i,j)는 n×m 행렬로써 프로세스와 자원간의 할당, 대기에 관한 상태 및 소속 값을 가지며 열(Row)은 프로세스를 행(Column)은 자원을 표시한다. *History*(i,j) = 1 (i=1,...,n, j=1,...,m)의 의미는 자원 R_j가 프로세스 P_i에 할당되었음을 나타낸다. *History*(i,j)에는 각각 한 개의 열과 행이 추가되어 사용되는데 *History*(n+1,j)는 자원 j의 소속 값을, *History*(i,m+1)에는 프로세스 i를 대기상태로 만든 자원의 정보가 들어 있다.

② 알고리즘의 이해도와 서술의 편의를 위해 두개의 집합변수 *child*와 *dsdnt*를 사용하였다.

```
Algorithm Request(i,j) // 루트 i가 자원 j를 요청 //
 1: begin
 2:   Key := History(n+1,j);
 3:   case
 4:    Key = null: // j가 가용자원일 경우 //
            History(i,j) := 1;
            History(n+1,j) := i;
 5:    Key = i : // 교착상태 판단 //
 6:    Key ≠ i : //단순 대기의 경우 //
 7:          History(i,m+1) := j;
 8:          for k = 1 to m do
               begin
                 if History(n+1,k) = i then
 9:                 History(n+1,k) := History(n+1,j);
                    // 소속 값을 i에서 자원 j의 소속
                    값으로 변경 //
                 endif
               end
             endfor
      endcase
 10:End
```

Request 알고리즘은 그림 2의 세 가지 상황을 CASE로 나누어서 실행한다. 명령문 4는 그림 2의 (a)에서 설명한바와 같이 교착상태가 아님이 분명하고, 명령문 6은 그림 2의 (b)와 같은 상황이다. 이 경우 두 개의 트리가 합쳐지게 되며 최악의 경우 O(m)의 시간 복잡도를 가지게 됨을 알 수 있으나, 단순히 대기상태가 되는 것으로써 두개의 트리를 합치는 작업은 여분의 프로세서로 다른 프로세스들과 동시에 실행시켜도 될 것이다. 명령문 5의 경우가 교착상태로 판단되는 경우인데, 이때 요구되는 시간이 O(1)이 되며 이 점이 본 논문의 핵심이다.

```
Algorithm Release(i,j) // 루트 i가 자원 j를 반납 //
 1: begin
 2:  for k = 1 to n do
      begin
        if History(k,m+1) = j then
 3:       child := History(k,m+1);
          // child는 자원 j에 의해 대기된 프로세
          스들을 포함 //
        endif
      end
    endfor
 4:  case
 5:   child = null: // j는 잎(leaf) 노드 //
          History(i,j) := 0;
          History(n+1,j) := null;
 6:   child = 1 : // 그림 3의 (b) 경우 //
          call adjust(i,j,k);
          call split(i,j,k);
 7:   otherwise : // 그림 3의 (c) 경우 //
          call adjust(i,j,k);
          call split(i,j,k);
      endcase
 8: End
```

```
Algorithm Adjust(i,j,k) // 반납시 트리의 분리를 위한
사전 작업 //
 1: begin
 2:   History(i,j) := 0;
      History(k,j) := 1;
      History(n+1,j) := k;
      History(k,m+1) := null;
 3: end
```

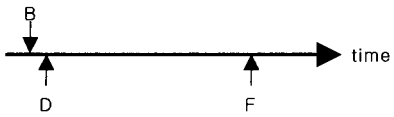
```
Algorithm Split(i,j,k) // 트리의 분리 //
 1: begin
 2:   Find all descendants resource nodes from
      process k and put them into dsdnt.
 3:   for all nodes in dsdnt do
        set their (n+1)th row to k;
      endfor
 4: end
```

자원의 반납은 교착상태를 발생시키지 않는다는 사실은 자명하다. *Adjust* 알고리즘은 *split*작업을 위한 사전 정지 작업으로 필요한 값들을 행렬의 해당란에 기입하며 O(1)의 작업임을 알 수 있다. *Release* 알고리즘에서 요구되는 시간은 *split* 알고리즘의 명령문 2로부터 기인하는데, 이 작업은 잘 알려진 트리 탐색(Tree traversal), 예를 들어 깊이 우선 탐색(Depth first search) 알고리즘을 사용하면 트리에 존재하는 에지의 개수에 비례하게 되고 본 논문에서 소개된 트리의 경우 O(n+m)의 시간이 소요된다.

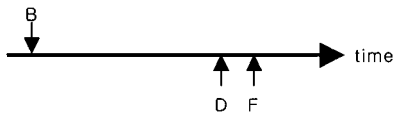
제안된 방법은 교착상태의 처리를 위해 전체적으로는 기존의 방법과 같은 O(n+m)의 시간을 요한다. 하지만 교착상태의 발견에는 O(1)의 시간이, 이후의 과정 즉, 대기상태를 위한 트리의 합병이나 반납을 위한 분리의 처리에 O(n+m)의 시간이 요구된다.

그림 4에서 B는 프로세스가 대기되는 시기이고 D는 교착상태의 발생 여부가 확인되는 시점이며 F는 한번의 교착상태 처리를 위해 필요한 모든 시간이 완료되는 시점이 된다. 따라서 [B, F]의 구간이 O(n+m)임을 알 수 있으며 (a)의 [B, D] 구간과 (b)의 [D, F] 구간이 상수 시간이 요구되는 구간이다. (a)의 [D, F] 구간은 이 후의 교착상태 발견을 O(1)으로 하기 위해 필요한 자료구조의 조정에 사용되는 시간이며 (b)의 [B, D] 구간은 사이클의 발생 유무를 판단하기 위해 RAG를 에지를 따라 탐색하는데 걸리는 시간이다.

언급한 바와 같이 대기상태로의 전환이나 자원의 반납과 같은 경우는 교착상태가 아님이 알려진 상황에서 이런 일들을, 다중처리 시스템에서 제공 가능한 여분의 프로세서를 활용하여, 일반 프로세스의 정상적인 진행과 병행하여 수행할 수 있을 것이며 이것은 곧 병렬성의



(a) 제안된 방법



(b) 기존 방법

그림 4 제안된 방법과 기존 방법의 수행 시간 비교

증가를 의미한다. 또한, 교착상태의 즉각적인 판단은 운영체제의 빠른 조치를 유도하여 자원의 활용도(Utilization)를 높이고, 교착상태에 관한한 최대한의 빠른 조치가 요구되는 실시간 시스템에도 매우 유용할 것이다.

5. 결 론

본 논문에서는 즉시 할당 상태와 함께 단일 자원, 단일 요청의 가정 하에서 시스템에 존재하는 프로세스와 자원의 개수를 각각 n과 m으로 할 때, 교착상태의 발견은 O(1)의 시간으로 대기와 반납을 위한 자료구조의 조정에 O(n+m)의 시간이 소요되는 알고리즘을 제시하였다. 이러한 O(n+m)의 시간은 교착상태가 아닌 경우의 작업으로 다중처리 환경에서 여분의 프로세서로 병행적 수행이 가능하다. 제안된 방법은 프로세스와 자원의 개수에 무관하게 교착상태의 발생을 즉각 판단할 수 있으므로 예측 가능한 시간에 운영체제로부터의 빠른 조치를 기대할 수 있으며 이 점은 실시간 환경에도 유효하다. 교착상태의 복구는 또 다른 문제이기는 하나 제안된 방법에서는 교착상태를 발생시키는 프로세스를 바로 알 수 있으므로 이 프로세스를 복구의 희생양(Victim)으로 선택함으로써 차선의 복구전략을 세울 수도 있을 것이다. 본 연구를 다중처리용 운영체제의 교착상태 처리 부분에 임베딩(Imbedding)하여 실제로 운영할 경우 성능에 관한 보다 많은 비교, 분석 자료를 취할 수 있을 것으로 예상하며 이 부분은 제반 여건의 구축을 전제로 향 후 연구로 남긴다.

참 고 문 헌

[1] W. S. Davis, T. M. Rajkumar, Operating Systems - A Systematic View, 5th Ed., Addison-Wesley, 2001.

[2] H. M. Deitel, D. R. Choffnes, Operating Systems, 3rd Ed., Prentice-Hall, 2004.

[3] M. Maekawa, A. E. Oldehoeft and R. R. Oldehoeft, Operating Systems - Advanced concepts, Benjamin-Cummings Pub., 1987.

[4] R. C. Holt, "Some Deadlock Properties of Computer Systems," ACM Computing surveys, Vol. 4, No. 3, Sep., 1972.

[5] J. G. Kim, "A Non-blocking Deadlock Detection Scheme for Multiprocessor Systems," Ph.D Thesis, SNU, Seoul, Feb., 1992.

[6] J. G. kim, "An Algorithmic Approach on Deadlock Detection for Enhanced parallelism in Multiprocessing Systems," Proc. of the 2nd Aizu int'l Symp. on Parallel Algorithms/Architecture Synthesis, IEEE Computer Society Press PR07870, pp. 233-238, Mar., 1997.

[7] 김 주균, 고 건, "다중처리 시스템하의 비 지연적 노트 발견 기법", 정보과학회논문지, 제18권 제5호, pp. 534-541, 1991.

[8] A. Shoshani and E. G. Coffman, "Prevention Detection and Recovery from System Deadlock," Proc. 4th annual Princeton Conf. on Information Sciences and System, Mar., 1970.

[9] T. F. Leibfried Jr., "A Deadlock Detection and Recovery Algorithm Using the Formalism of a Directed Graph Matrix," Operating System Review, Vol. 23, No. 2, Apr., 1989.

[10] S. S. Isloor and T. A. Marsland, "The Deadlock Problem: An Overview," IEEE Computer, Sep., 1980.

[11] Y. S. Ryu and K. Koh, "A Predictable Deadlock Detection Technique for a Single resource and single Request System," Proc. of the 14th IASTED Int'l Conf. on Applied Informatics, pp. 35-38, Innsbruck, Austria, Feb., 1996.

[12] J. G. kim and K. Koh, "An O(1) Time Deadlock Detection Scheme in Single Unit and Single Request Multiprocessor System," IEEE TENCON'91, Vol. 2, pp. 219-223, aug., 1991.

김 주 균

1985년 서울대학교 계산통계학과 졸업
1985년~1986년 DEC Korea 연구원
1988년 서울대학교 계산통계학과 계산학 석사. 1992년 서울대학교 계산통계학과 계산학 박사. 1992년~현재 숙명여대 정보과학부 컴퓨터과학전공 교수. 관심분야는 (분산, 병렬)운영체제, 성능평가, Web caching, Embedded system