Stack Resource Policy를 사용하는 동적 우선순위 스케줄링에서 작업 큐잉을 위한 효율적인 자료구조

(An Efficient Data Structure for Queuing Jobs in Dynamic Priority Scheduling under the Stack Resource Policy)

한 상 철 † 박 문 주 †† 조 유 근 †††
(Sangchul Han) (Moonju Park) (Yookun Cho)

요 약 Stack Resource Policy (SRP)는 몇 가지 독특한 특성을 가진 실시간 동기화 프로토콜이다. 그 특성 중의 하나는 초기 수행 차단(early blocking)으로서, 공유자원을 요청하는 시점에 작업의 수행을 정지시키는 것이 아니라, 작업의 수행 시점 자체를 연기하도록 한다. SRP가 EDF와 같은 동적 우선순위 스케줄링 알고리즘과 같이 사용될 경우, 초기 수행 차단을 지원하기 위해 스케줄러는 수행이 블록(block)되지 않을 작업 중 가장 우선순위가 높은 작업을 선택해야 하며, 이러한 탐색 연산은 수행 오버헤드(runtime overhead)의 원인이 된다. 본 논문에서는 SRP와 EDF를 같이 사용할 때의 스케줄러 수행 오버헤드를 분석한다. 기존의 준비 큐(ready queue) 구현 방식과 탐색 알고리즘을 사용하면 작업의 수가 많아짐에 따라 작업 탐색 오버헤드가 매우 커진다. 이 문제를 해결하기 위하여, 본 논문은 스케줄러가 효율적으로 작업을 탐색할 수 있는 준비 큐 자료구조와 $O(\lceil \log_2 n \rceil)$의 복잡도를 가지는 작업 탐색 알고리즘을 제안한다.

키워드 : 실시간 스케줄링, 동적 우선순위, EDF, 자원 공유, SRP, 준비 큐, 이진트리

Abstract The Stack Resource Policy (SRP) is a real-time synchronization protocol with some distinct properties. One of such properties is early blocking; the execution of a job is delayed instead of being blocked when requesting shared resources. If SRP is used with dynamic priority scheduling such as Earliest Deadline First (EDF), the early blocking requires that a scheduler should select the highest-priority job among the jobs that will not be blocked, incurring runtime overhead. In this paper, we analyze the runtime overhead of EDF scheduling when SRP is used. We find out that the overhead of job search using the conventional implementations of ready queue and job search algorithms becomes serious as the number of jobs increases. To solve this problem, we propose an alternative data structure for the ready queue and an efficient job-search algorithm with $O(\lceil \log_2 n \rceil)$ time complexity.

Key words : real-time scheduling, dynamic priority, EDF, resource sharing, SRP, ready queue, binary tree

1. 서 론

실시간 환경에서 작업(job)들이 자원을 공유하면 접근 제어(access control)가 필요하다. 만약 자원에 대한 접근을 제한 없이 허용하면, 교착상태(deadlock) 또는 다중 우선순위 역전(multiple priority inversion) 현상이 발생할 수 있다. 교착상태에 빠진 작업은 교착상태 탐지 장치(deadlock detection mechanism)가 없으면 더 이상 수행을 진행할 수 없으며, 다중 우선순위 역전 현상이 발생하면 작업의 차단 시간(blocking time)을 예측할 수 없어서 작업의 시간제약 충족을 보장할 수 없다.

실시간 환경에서의 공유자원 접근 제어 문제를 해결하기 위하여 많은 연구들이 수행되었다. Sha와 Raj-

· 본 연구는 두뇌한국21 사업의 지원으로 수행되었음
· † 학생회원 : 서울대학교 컴퓨터공학과
schan@ssrnet.snu.ac.kr
†† 비 회 원 : IBM 유비쿼터스 컴퓨팅 연구소
mjupark@kr.ibm.com
††† 종신회원 : 서울대학교 컴퓨터공학부 교수
cho@cse.snu.ac.kr
논문접수 : 2004년 7월 16일
심사완료 : 2006년 3월 28일

kuma, Lehoczky 등은 고정 우선순위(fixed priority) 스케줄링에서 자원 공유 문제를 해결하기 위해 Priority Inheritance Protocol과 Priority Ceiling Protocol(PCP)를 제안하였다[1]. Chen과 Lin은 PCP를 EDF[2] 스케줄링에서 사용할 수 있도록 확장하였으며[3], Jeffay는 작업이 공유자원을 사용하는 동안 불필요한 선점(preemption)을 방지하기 위해 작업의 마감시간(deadline)을 변경하는 Dynamic Deadline Modification을 제안하였다[4].

Baker는 Stack Resource Policy(SRP)를 제안하였으며[5], SRP는 다음과 같은 몇 가지 독특한 특성을 가지고 있다. 첫째, SRP는 고정 우선순위 스케줄링과 동적 우선순위(dynamic priority) 스케줄링 모두에서 사용될 수 있다. 이것은 SRP가 우선순위(priority)와 선점순위(preemption level)를 구별하기 때문에 가능하다. 둘째, 일단 작업이 수행을 시작하였다면 그 작업은 공유자원을 요청할 때 블록(block)되지 않으며, 대신에 작업 수행의 시작이 연기된다. 이것을 초기 수행 차단(early blocking)이라고 한다. 셋째, SRP는 한 종류에 여러 개의 자원이 있는 경우에도 사용될 수 있다.

우리는 초기 수행 차단이 EDF와 같은 동적 우선순위 스케줄링의 수행 오버헤드(runtime overhead)에 미치는 영향을 연구하였다. 어떤 작업이 공유자원을 접근할 때 블록될 것으로 예상되면, SRP에서는 그 작업이 준비된 작업(ready job)들 중에서 가장 우선순위가 높은 경우에도 수행시키지 않아야 한다. 따라서 매 스케줄링 시점마다, 스케줄러는 블록되지 않을 작업 중에서 가장 우선순위가 높은 작업을 선택해야 하며, 그러한 작업을 최적 작업(the most eligible job)이라고 부른다. 이 과정은 작업들을 효율적으로 관리하지 않으면 상당한 시간을 소비한다. 최악의 경우에 스케줄러는 모든 준비된 작업을 검사해야하며, 작업의 수가 많은 고성능 실시간 시스템에서는 심각한 수행 오버헤드가 발생한다.

본 논문에서는 SRP를 사용할 때 준비 큐의 구현 방식에 따른 EDF 스케줄링의 수행 오버헤드를 분석한다. 기존의 구현방식을 사용하면 작업을 탐색하는 오버헤드는 작업의 수가 증가함에 따라 선형으로 증가한다. 본 논문에서는 준비 큐를 위한 대안의 자료구조를 제안하고, 최적 작업을 $O(\lceil \log_2 n \rceil)$의 복잡도로 탐색하는 알고리즘을 제시한다.

본 논문의 구성은 다음과 같다. 2장에서는 SRP의 개관을 설명하고 SRP가 사용될 때의 EDF 스케줄링의 수행 오버헤드를 분석한다. 3장에서는 준비 큐를 위한 자료구조와 작업을 선택하는 알고리즘을 제안한다. 4장에서는 실험을 통하여 제안된 기법을 기존의 기법과 비교, 평가한다. 마지막으로 5장에서 결론을 맺는다.

2. Stack Resource Policy

SRP는 실시간 작업의 공유자원 접근을 제어하는 동기화 프로토콜(synchronization protocol)으로서 선점순위의 개념을 도입하였다. 모든 작업 J는 정적인 선점순위 $\pi(J)$를 부여 받는다. 작업 J_i는 $\pi(J_i)$가 $\pi(J_j)$보다 높을 때에만 J_j를 선점할 수 있으며, 이 조건을 만족하는 어떤 인자도 선점순위로 사용될 수 있다. 예를 들어, EDF 스케줄링에서는 상대적 마감시간(relative deadline)에 따라 작업의 선점순위를 부여할 수 있다. 즉, D_k가 작업 J_k의 상대적 마감시간일 때, 선점순위는 다음과 같이 정의된다.

$$\pi(J_i) > \pi(J_j) \Leftrightarrow D_i < D_j$$

자원의 상한(resource ceiling)은 그 자원을 사용하는 작업들의 선점순위 중 가장 높은 값으로 정의되며, 시스템의 상한(system ceiling)은 현재 사용 중인 자원들의 상한 중 가장 높은 값으로 정의된다. SRP는 작업의 선점순위가 시스템 상한보다 높을 때에만 그 작업이 수행을 시작할 수 있도록 허용한다. 작업의 선점순위가 현재 시스템 상한보다 높지 않으면, 스케줄러는 시스템 상한이 그 작업의 선점순위보다 낮아질 때까지 그 작업의 시작을 연기하며, 이를 초기 수행 차단이라고 한다. 수행 차단이 해제된 후 수행을 시작한 작업은 블록되지 않으며, 교착상태가 발생하지 않음이 보장된다.

초기 수행 차단은 SRP의 중요한 특성이다. 이 특성으로 인하여 작업의 우선순위 조정이나 블록이 필요 없기 때문에 SRP의 공유자원 접근 과정은 PCP에 비해 단순하며, 블록으로 인해 발생하는 문맥교환도 없다. 그러나 초기 수행 차단을 지원하기 위해 스케줄러는 작업 탐색의 오버헤드를 감수해야 한다. 스케줄러는 선점순위가 현재 시스템 상한보다 높은 작업 중에서 우선순위가 가장 높은 작업, 즉 최적 작업을 찾아야 한다.

고정 우선순위 스케줄링에서는 이러한 제한이 스케줄러에게 거의 영향을 주지 않는다. 왜냐하면, 고정 우선순위 스케줄링에서는 선점순위를 우선순위와 동일하게 줄 수 있으며, 따라서 우선순위가 가장 높은 작업이 선점순위도 가장 높기 때문이다. 스케줄러는 우선순위가 가장 높은 작업의 선점순위가 시스템 상한보다 더 높은가만 검사하면 된다.

동적 우선순위 스케줄링에서는 우선순위가 가장 높은 작업의 선점순위가 시스템 상한보다 낮을 수 있으며, 반대로 우선순위가 비교적 낮은 작업의 선점순위가 시스템 상한보다 높을 수도 있다. 이러한 경우, 최적 작업을 찾는 과정이 스케줄러의 오버헤드가 되며 전체 작업의 수가 많을수록 오버헤드가 증가한다.

표 1의 작업 집합은 EDF 스케줄링에서 SRP가 사용될 때(이후로 EDF+SRP로 표기한다) 최적 작업을 찾

는 과정이 스케줄러에 미치는 영향을 보여준다. 작업의 선점순위는 상대적 마감시간에 따라 부여되었다. 즉, 상대적 마감시간이 작을수록 선점순위가 높다. 자원 R_1은 J_1만이 사용하고, R_2는 J_3과 J_8이 공유한다. R_3는 나머지 작업들이 공유한다. R_1과 R_2, R_3의 자원 상한은 각각 8과 6, 7이다. 이 작업 집합은 [5]의 정리 10에 의하여 EDF+SRP로 스케줄 가능하다. 그림 1은 이 작업 집합에 대한 EDF+SRP 스케줄이다. 각 작업의 도착은 ↑로 표시되고, 절대적 마감시간(absolute deadline)은 ↓로 표시되며, 임계구역(critical section)은 채운 사각형으로 표시된다. 시간 0에 J_8이 도착하고 수행을 시작한다. 시간 2에 J_8은 R_2에 락(lock)을 설정하고 임계구역으로 들어간다. 이 때 시스템 상한은 R_2의 자원 상한인 6이 된다. J_4와 J_5, J_6, J_7은 J_8이 임계구역을 수행하는 동안 도착하며, 선점순위가 시스템 상한인 6보다 작으므로 수행을 시작하지 못한다. 시간 8에 J_1이 도착하고 J_8을 선점한다. J_1의 선점순위는 8이므로 시스템 상한보다 높다. 시간 10과 12에 각각 J_3과 J_2가 도착하지만 마감시간이 J_1보다 늦으므로 J_1을 선점하지 못한다. 시간 12에 J_1은 R_1에 락을 걸고 임계구역으로 들어간다. 이 때, 시스템 상한은 8로 높아진다. 시간 13에 J_1은 R_1의 락을 해제(unlock)하고 수행을 마친다. 이 때 시스템 상한은 6으로 복원되며, 스케줄러는 준비된 작업 J_2, J_3, ... , J_7 중에서 최적 작업, 즉, 선점순위가 시스템 상한인 6보다 크며 마감시간이 가장 빠른 작업을 탐색해야 한다.

준비된 작업들은 준비 큐에 삽입되어 관리된다. 일반적으로 준비 큐는 정렬 리스트(sorted list) 또는 정렬 힙(sorted heap)을 사용하여 구현한다. 두 자료구조에서 작업들은 (절대적) 마감시간에 따라 정렬된다. 정렬 리스트의 삽입연산(insertion operation)은 $O(n)$의 복잡도를 가지며, 정렬 힙의 삽입연산은 $O(\log_2 n)$의 복잡도를 가진다. 단, n은 작업의 수이다. 두 자료구조에서 마감시간이 가장 빠른 작업은 항상 헤드(head) 또는 루트(root)에 위치하기 때문에, 마감시간이 가장 빠른 작업을 탐색하는 연산은 $O(1)$의 복잡도를 가진다.

그러나 공유자원을 통제하기 위해 SRP를 사용하면 그렇지 않다. 헤드 또는 루트에 위치한 작업의 선점순위가 시스템 상한보다 작으면 스케줄러는 최적 작업을 선택하기 위해 다른 작업들을 검사해야 한다. 그림 2는 위의 예에서 시간이 13일 때 준비 큐의 상태를 보여준다 (J_8은 스택으로 옮겨졌다고 가정한다. SRP의 기본적 구현 방법은 [5]를 참조할 수 있다). 시간 13에 시스템 상한은 6이지만 헤드(또는 루트)에 위치한 J_7의 선점순위는 2이다. 스케줄러는 선점순위가 6보다 큰 작업을 찾기 위해 준비큐에 있는 각 작업들을 하나씩 검사해야 하며, 마지막에 최적 작업인 J_2를 발견한다. 그러므로 기존의 준비큐 구현방식으로 최적 작업을 찾는 탐색연산(search operation)의 복잡도는 최악의 경우 $O(n)$이다. 이 오버헤드는 작업의 수가 증가함에 따라 크게 증가하며, 시스템 자원을 낭비하게 된다.

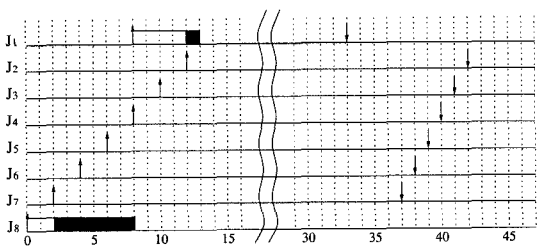
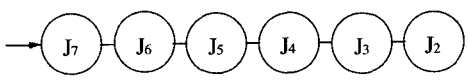
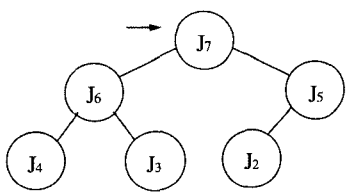


그림 1 EDF+SRP에 의한 스케줄



(a) sorted list



(b) sorted heap

그림 2 준비 큐의 자료구조

표 1 작업 집합의 예. r_i는 도착시간, ξ_i는 임계구역의 길이, C_i는 최악의 수행시간

J_i	r_i	D_i	$\pi(J_i)$	C_i	ξ_i	resource
J_1	8	25	8	5	1	R_1
J_2	12	30	7	2	1	R_3
J_3	10	31	6	3	1	R_2
J_4	8	32	5	2	1	R_3
J_5	6	33	4	3	1	R_3
J_6	4	34	3	2	1	R_3
J_7	2	35	2	3	1	R_3
J_8	0	100	1	10	6	R_2

3. 효율적인 작업 관리 기법

EDF+SRP 스케줄링에서 스케줄러는 최적 작업을 찾기 위해 작업의 마감시간과 선점순위를 모두 참조한다. 따라서 최적 작업을 효율적으로 찾기 위해서는 작업들이 마감시간의 순서뿐만 아니라 선점순위의 순서로도 정렬되어야 한다. 본 장에서는 완전이진트리(complete binary tree)를 사용하여 마감시간과 선점순위의 순서로 작업을 정렬하는 준비 큐 구현방식과 효율적으로 최적 작업을 찾는 알고리즘을 제안한다.

3.1 효율적인 작업 선택을 위한 자료구조

본 논문에서 제안하는 작업 관리 기법은 준비 큐의 자료구조로서 완전이진트리를 사용한다. 그림 3은 완전이진트리를 사용한 준비 큐의 예이다. 트리의 단말노드(leaf node)들은 선점순위와 연관되며, 선점순위가 p인 작업은 선점순위 p와 연관된 단말노드에 위치한다(작업들의 선점순위는 서로 다르다고 가정한다). 선점순위가 p인 어떤 작업이 도착하면 그 작업의 TCB(Task Control Block)에 대한 포인터가 선점순위 p와 연관된 단말노드에 기록된다. 현재 선점순위가 p인 준비된 작업이 없으면 유휴태스크(idle task)의 TCB에 대한 포인터가 선점순위 p와 연관된 단말노드에 기록된다. 각 내부노드(internal node)는 그 노드의 두 자식노드(child node)가 가리키는 작업 중에 마감시간이 빠른 작업의 TCB에 대한 포인터를 복사하여 기록한다. 따라서 루트노드(root node)는 트리 전체에서 마감시간이 가장 빠른 작업의 TCB에 대한 포인터를 갖는다. 그림 3에서 각 노드의 숫자는 그 노드가 가리키는 작업의 절대적 마감시간(그 작업의 TCB에 기록되어 있다)이며, **D(node)**로 표현된다. 유휴태스크의 마감시간은 ∞으로 정의된다. 또한, 각 노드는 **FLAG(node)**로 표현되는 플래그를 가지고 있다. 어떤 노드가 부모노드(parent node)의 왼쪽 자식노드이면 **FLAG(node)**는 LEFT이며, 오른쪽 자식노드이면 **FLAG(node)**는 RIGHT이다. 루트노드의 **FLAG(node)**는 ROOT이다.

초기에 모든 노드들은 유휴태스크의 TCB에 대한 포인터를 갖는다. 어떤 작업이 도착하면 그 작업의 TCB에 대한 포인터가 그 작업의 선점순위와 연관된 단말노드에 기록된다. 그 후에 루트노드에 이르는 중간노드들의 TCB 포인터가 갱신된다. 마감시간이 가장 빠른 작업이 도착할 때가 최악의 경우이며, 이 경우에 단말노드부터 루트노드까지의 경로에 있는 모든 노드의 TCB 포인터가 갱신되어야 한다. 마찬가지로, 어떤 작업이 종료하면 그 작업의 선점순위와 연관된 단말노드에 유휴태스크의 TCB에 대한 포인터가 기록되며, 루트노드에 이르는 중간노드들의 TCB 포인터가 갱신된다. 마감시간이 가장 빠른 작업이 종료할 때가 최악의 경우이며, 단말노드부터 루트노드까지의 경로에 있는 모든 노드의 TCB 포인터가 갱신되어야 한다.

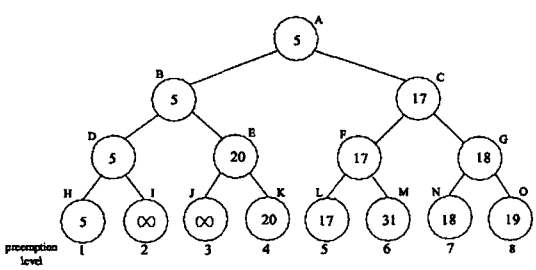


그림 3 준비큐로서의 완전이진트리

작업의 수가 n개일 때, 시스템이 제공하는 선점순위의 수 m은 적어도 n개 이상이어야 한다. 제안한 자료구조는 완전이진트리이므로 $m=2^k$ ($2^{k-1}<n\le 2^k$)이다. $k=\lceil \log_2 n \rceil$ 이므로, $\log_2 m=\log_2 2^{\lceil \log_2 n \rceil}=\lceil \log_2 n \rceil$. 따라서 트리의 높이는 $\log_2 m+1=\lceil \log_2 n \rceil+1$이며, 삽입연산과 삭제연산의 복잡도는 $O(\lceil \log_2 n \rceil)$이다.

3.2 작업 탐색 알고리즘

그림 4는 완전이진트리를 탐색하여 최적 작업을 찾는 트리탐색 알고리즘이다. 이 알고리즘은 주어진 시스템 상한 S보다 큰 선점순위를 갖는 작업 중에서 마감시간이 가장 빠른 작업을 찾는다. 이 알고리즘은 높이가 1이며 루트노드가 단말노드인 부분트리(sub-tree)부터 탐색을 시작한다. 부분트리가 전체 완전이진트리가 될 때까지 부분트리의 높이를 점차 증가시키면서 부분트리 안에서 최적 작업을 찾는다. 부분트리의 루트노드는 변수 *subroot*로 표현되며, 그 부분트리 안의 최적 작업은 변수 *candidate*로 표현된다.

초기에 *subroot*와 *candidate*는 모두 선점순위 $(S+1)$와 연관된 단말노드를 가리킨다. 선점순위는 정적으로(static) 연관되어 있으므로 이 노드를 찾는 과정의 복잡도는 $O(1)$이다. *subroot*가 부모노드의 왼쪽 자식노드일 경우에 변수 *competitor*는 *subroot*의 형제노드(sibling node)를 가리킨다. *competitor*가 루트노드인 부분트리를 T라고 하자. T에 속한 모든 노드는 선점순위가 S보다 크며, *competitor*는 T에 속한 모든 노드 중에서 마감시간이 가장 빠른 작업의 TCB를 가지고 있다. 따라서 *subroot*와 *candidate*의 부모노드가 루트노드인 부분트리 T^* 안에서 최적 작업은 *candidate*와 *competitor* 중에서 마감시간이 빠른 작업이다. *subroot*가 부모노드의 오른쪽 자식노드일 경우에 *subroot*의 형제노드를 고려할 필요가 없다. 왜냐하면 *subroot*의 형제노드가 루트노드인 부분트리에 속한 모든 노드는 선점순위가 S보다 작기 때문이다. 트리탐색 알고리즘은 최악

의 경우에 $3\lceil \log_2 n \rceil$ 회의 비교연산과 $(3\lceil \log_2 n \rceil + 2)$회의 대입연산을 수행한다.

```
1:    candidate := leaf node associated with preemption level S+1
2:    subroot := candidate
3:    while FLAG(subroot) != ROOT do
4:          if FLAG(subroot) == LEFT then
5:                competitor := sibling of subroot
6:                if D(competitor) < D(candidate) then
7:                      candidate := competitor
8:                endif
9:          endif
10:         subroot := paraent node of subroot
11:   done
12:   return TCB(candidate)
```

그림 4 트리탐색 알고리즘

트리탐색 알고리즘의 효율성은 시스템 상한 S의 영향을 받는다. S가 충분히 크면, 선점순위가 S보다 큰 단말노드들을 선형탐색(linear search) 하는 것이 더 효율적이다. 완전이진트리를 이용한 준비 큐의 선형탐색 알고리즘을 그림 5에 기술하였다. 이 알고리즘은 $3(S_{\max}-S-1)+2$회의 대입연산과 $2(S_{\max}-S-1)+1$회의 비교연산을 수행한다.

주어진 시스템 상한 S에 대해 트리탐색 알고리즘과 선형탐색 알고리즘의 효율성 비교는 다음과 같이 할 수 있다. δ_a을 대입연산 수행시간이라고 하고 δ_c을 비교연산 수행시간이라고 하자. 트리탐색 알고리즘의 오버헤드는 $\Delta_T = 3\lceil \log_2 n \rceil \delta_c + (3\lceil \log_2 n \rceil + 2)\delta_a$이고, 선형탐색 알고리즘의 오버헤드는 $\Delta_L = (2(S_{\max}-S-1)+1)\delta_c + (3(S_{\max}-S-1)+2)\delta_a$이다. 시스템이 1부터 64까지 64개의 선점순위를 지원하고 $(S_{\max}=m=64)$, $\delta_c=\delta_a$라고 하자. 그러면, $S>56$ 일 때, $\Delta_L < \Delta_T$ 이다. 즉, 시스템 상한이 56보다 크면 선형탐색 알고리즘을 적용하는 편이 더 효율적이다. 본 논문에서 제안한 자료구조는 조건에 적합한 탐색 알고리즘을 사용함으로써 최적 작업 탐색연산 오버헤드를 최소화할 수 있다.

```
1:    candidate := leaf node associated with preemption level S+1
2:    s := S + 2
3:    while s ≤ Smax do
4:          competitor := leaf node associated with preemption level s
5:          if D(competitor) < D(candidate) then
6:                candidate := competitor
7:          endif
8:          s := s + 1
9:    done
10:   return TCB(candidate)
```

그림 5 선형탐색 알고리즘

3.3 동일 선점순위와 동적 작업집합 문제

2장에서 언급한 바와 같이 EDF+SRP 스케줄링에서 전형적인 선점순위 부여 방식은 상대적 마감시간에 따른 부여 방식이다. 만약 여러 작업들의 상대적 마감시간이 동일하면 이 작업들은 동일한 선점순위를 부여 받는다. 그러나 3.1절에서 기술한 완전이진트리를 이용한 준비 큐는 하나의 선점순위에 여러 작업을 수용할 수 없다. 이 문제를 해결하기 위해 CRGM(Constant Ratio Grid Method)을 사용할 수 있다[6,7]. 각 선점순위마다 FIFO 큐를 할당하고, FIFO 큐의 헤드에 위치하는 작업을 완전이진트리의 단말노드가 가리키도록 한다. i번째 FIFO 큐는 $[D^i_{\min}, D^i_{\max})$ 범위의 상대적 마감시간을 갖는 작업이 삽입되며, 그리드 비율(grid ratio)은 $r^i = D^i_{\max}/D^i_{\min}$ 이다. 태스크 집합이 $\Sigma(C_i/T_i) \le 1/r$을 만족하면 스케줄 가능하다. CRGM은 선점순위 또는 우선순위의 수가 제한적인 시스템에서 약간의 이용률(utilization) 손해를 감수하고 많은 수의 선점순위 또는 우선순위를 지원할 수 있게 한다[8]. 예를 들어, 시스템이 256개의 선점순위를 지원하고, 최소 상대적 마감시간이 10μs이며 최대 상대적 마감시간이 1s라고 하자. 그러면 그리드 비율은 $\sqrt[256]{100000} \approx 1.046$이고, 이용률 한계(utilization bound)는 $1/r \approx 95.60\%$ 이다.

본 논문에서는 작업집합이 정적이라고 가정한다. 정적 작업집합은 작업의 수와 특성(수행시간, 마감시간 등)이 시간에 따라 변하지 않으며, 따라서, 선점순위도 정적으로 결정된다. 만약, 작업집합이 동적이라면 CRGM을 사용하여 작업의 도착이 동적인 상황에서도 정적인 선점순위를 부여할 수 있다. 각 작업의 선점순위는 미리 정해지지 않고, 작업이 도착할 때 상대적 마감시간에 따라 선점순위를 부여한다. 이 경우에 시스템이 제공하는 선점순위의 수 m은 작업의 수 n과 관계없이 결정되며, 삽입연산과 삭제연산의 복잡도는 $O(\log_2 m)$이다.

4. 실 험

본 장에서는 제안한 기법의 효율성을 검증하기 위하여 주기 태스크집합에 대한 스케줄러 오버헤드를 기존의 구현기법과 비교한다. 고려한 자료구조는 정렬 리스트와 비정렬 리스트(unsorted list), 정렬 힙, 완전이진트리이다. 정렬 리스트와 비정렬 리스트는 이중 연결 리스트(doubly linked list)를 이용하여 구현하였으며, 정렬 힙과 완전이진트리는 배열(array)을 이용하여 구현하였다.

스케줄러의 주요 활동은 준비 큐에 작업 삽입, 수행할 작업 탐색, 준비 큐에서 작업 삭제 등이다. 이런 활동으로 인한 오버헤드를 각각 삽입 오버헤드(Δ_{ins}), 탐색 오버헤드(Δ_{srch}), 삭제 오버헤드(Δ_{del})라고 한다. 새로운 작

표 2 삽입, 삭제, 탐색 연산의 최악의 수행 오버헤드

	기존 구현 기법			제안한 기법
	정렬 리스트	비정렬 리스트	정렬 힙	완전이진트리
Δ_{ins}	$0.22n+1.16$	1.09	$0.52\lfloor \log_2 n \rfloor +0.77$	$0.38\lceil \log_2 n \rceil +1.00$
Δ_{del}	0.91	0.90	$0.59\lfloor \log_2 n \rfloor +0.89$	$0.38\lceil \log_2 n \rceil +1.10$
Δ_{sel}	$0.14n+0.56$	$0.27n+0.47$	$0.13n+1.05$	$0.34\lceil \log_2 n \rceil +0.90$

업이 도착하면 그 작업은 준비 큐에 삽입되고 몇몇 자료가 갱신된다. 이것은 삽입 오버헤드이다. 그 다음에 준비된 작업들 중에서 최적 작업을 탐색한다. 왜냐하면 새로 도착한 작업들 중에 우선순위가 현재 수행중인 작업보다 높은 작업이 있을 수 있기 때문이다. 따라서 탐색 오버헤드가 발생한다. 어떤 작업이 수행을 마치면 스케줄러는 그 작업을 준비 큐에서 제거하고, 다른 최적 작업을 탐색한다. 삭제 오버헤드와 탐색 오버헤드가 발생한다. 그러므로 운영체제의 다른 부분에서 발생하는 오버헤드를 무시한다면 태스크의 주기 당 전체 오버헤드는 $\Delta_{total}=\Delta_{ins}+\Delta_{del}+2\Delta_{srch}$이다[9].

본 실험에서는 펜티엄 120MHz 프로세서에서 작업의 수에 따른 각 자료구조의 삽입, 삭제, 탐색 연산의 최악 수행 오버헤드를 측정하고, 그 결과에 대해 Least Square Method를 적용하여 표 2의 수식을 유도하였다. 단위는 μs이고 n은 작업의 수이다. 작업의 선점순위는 서로 다르다고 가정하였다.

비정렬 리스트의 탐색 오버헤드는 정렬 리스트의 탐색 오버헤드보다 크다. 그 이유는 비정렬 리스트에서는 마감시간과 선점순위를 모두 비교하지만 정렬 리스트에서는 선점순위만을 비교하기 때문이다. 정렬 힙의 삽입, 삭제 오버헤드의 계수는 완전이진트리보다 다소 크다. 이것은 정렬 힙의 삽입, 삭제 연산은 부모노드와 자식노드의 교환을 수반하기 때문이다. 정렬 리스트와 비정렬 리스트, 정렬 힙의 경우 탐색 오버헤드의 복잡도는 모두 $O(n)$이다. 반면에 완전이진트리는 예상한 바와 같이 모든 연산의 복잡도가 $O(\lceil \log_2 n \rceil)$이다.

그림 6은 각 자료구조에 대한 주기 당 전체 오버헤드(Δ_{total})를 보인다. 그래프는 표 2의 결과를 사용하여 그렸다. 작업의 수가 15개 이하일 경우에는 정렬 리스트와 비정렬 리스트의 오버헤드가 가장 작고, 완전이진트리의 오버헤드는 이보다 다소 크다. 그러나 작업의 수가 15개 이상일 경우에는 완전이진트리의 오버헤드가 가장 작고, 작업의 수가 많을수록 그 격차는 점점 커진다.

5. 결 론

본 논문에서는 EDF+SRP 스케줄링의 수행 오버헤드를 분석하였다. 기존의 준비 큐 자료구조와 작업 탐색 알고리즘을 이용하면 최적 작업, 즉, 선점순위가 시스템

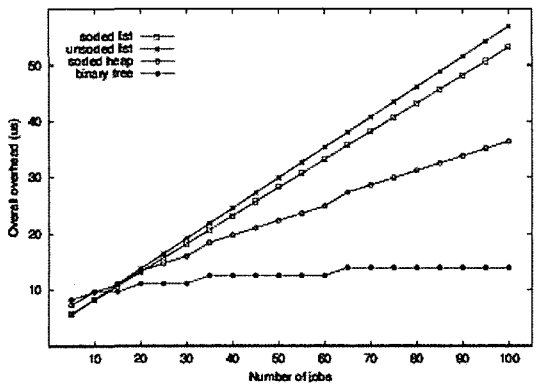


그림 6 주기 당 전체 오버헤드

상한보다 높고 마감시간이 가장 빠른 작업을 탐색하기 위한 스케줄러의 수행 오버헤드가 매우 커 질 수 있다. 리스트나 힙을 사용하여 준비 큐를 구현하면 최악의 경우 모든 준비된 작업을 검사해야 한다. 즉, 최적 작업을 탐색하는 연산의 복잡도는 $O(n)$이다.

본 논문은 EDF+SRP 스케줄링에서 작업을 효율적으로 관리할 수 있는 기법을 제안하였다. 이 기법은 완전이진트리를 사용하여 준비 큐를 구현한다. 준비된 작업들은 완전이진트리에서 마감시간과 선점순위를 모두 고려하여 정렬되고, 제시한 작업 탐색 알고리즘은 완전이진트리에서 최적 작업을 $O(\lceil \log_2 n \rceil)$의 복잡도로 찾을 수 있다. 실험 결과에 의하면 제안된 기법은 스케줄러의 전체 오버헤드를 상당히 감소시킨다.

참 고 문 헌

[1] L. Sha, R. Rajkumar, and J. Lehoczky, "Priority Inheritance Protocols: An Approach to Real-Time Synchronization," IEEE Transactions on Computers, Vol.39, No.9, pp.1175-1185, 1990.

[2] C. L. Liu and J. W. Layland, "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment," Journal of the ACM, Vol.20, No.1, pp.46-61, 1973.

[3] M. Chen and K. Lin, "Dynamic Priority Ceilings: A Concurrency Control Protocol for Real-Time Systems," Real-Time Systems, Vol.2, No.4, pp.325-346.

[4] K. Jeffay, "Scheduling Sporadic Tasks with Shared Resources in Hard-Real-Time Systems," In Proceedings of 13th IEEE Real-Time Systems Symposium, pp.89-99, 1992.
[5] T. Baker, "Stack-Based Scheduling of Real-Time Processes," Real-Time Systems, Vol.3, No.1, pp.67-100, 1991.
[6] J. P. Lehoczky and L. Sha, "Performance of Real-Time Bus Scheduling Algorithms," In Proceedings of the Joint Conference on Computer Performance Modelling, Measurement and Evaluation, pp.44-53, 1986.
[7] M. Park, L. Sha, and Y. Cho, "A Practical Approach to Earliest Deadline Scheduling," SNU-CE-TR-2001-2, 2001.
[8] J. W. Liu, Real-Time Systems, Prentice-Hall, 2000.
[9] K. M. Zuberi, P. Pillai, and K. G. Shin, "EMERALDS: A Small-Memory Real-Time Microkernel," In Proceedings of 17th ACM Symposium on Operating Systems Principles, pp.277-299, 1999.

한 상 철

1998년 연세대학교 컴퓨터과학과 졸업(공학사). 2000년 서울대학교 컴퓨터공학과 졸업(공학석사) 2000년~현재 서울대학교 컴퓨터공학과 박사과정. 관심분야는 실시간 시스템, 스케줄링 알고리즘

박 문 주

1996년 서울대학교 조선공학과 졸업(공학사). 1998년 서울대학교 컴퓨터공학과 졸업(공학석사). 2002년 서울대학교 컴퓨터공학과 졸업(공학박사). LG전자 단말연구소 책임연구원. 현재 IBM Ubiquitous Computing 연구소

조 유 근

1971년 서울대학교 건축공학과 학사 1978년 미네소타대학교 컴퓨터과학 박사 1979년~현재 서울대학교 컴퓨터공학부 교수. 1984년~1985년 미네소타대학교 교환 교수. 1993년~1995년 서울대학교 중앙교육연구전산원장. 1999년~2001년 서울대학교 공과대학 부학장. 2001년~2002년 한국정보과학회 회장. 관심분야는 운영체제, 알고리즘 설계 및 분석, 암호학