2006년도 한국정보과학회 가을 학술발표논문집 Vol. 33, No. 2(A)

선행스케줄링에서 배타적 자원접근

박학봉[O], 한상철, 김희헌, 박민규', 조성제[+], 조유근

서울대학교, 건국대학교, 단국대학교[+]

{hbpark, schan, hhkim, cho}@ssrnet.snu.ac.kr, minkyup@kku.ac.kr' sjcho@dankook.ac.kr[+]

Mutually Exclusive Resource Access in Pre-Scheduling

Xuefeng Piao[O], Shangchul Han, Heeheon Kim, Minkyu Park', Seongje Cho[+]', Yookun Cho

Seoul National University, KonKuk University', DanKook University[+]

요 약

선행스케줄링(pre-scheduling)은 정적인 작업(periodic job)과 동적인 작업(sporadic job)을 유연하게 처리하기 위해 제안된 스케줄링 방식이다. 이 방식은 오프라인 컴포넌트와 온라인 컴포넌트로 구성되며 오프라인 컴포넌트에서는 비주기적으로 도착하는 동적인 작업들을 고려하여 정적인 작업들을 여러 부분작업으로 분할하고, 그리고 각 부분작업들의 실행시간, 준비시간, 마감시간을 부여하고 실행순서를 결정한다. 온라인 컴포넌트에서는 이 정보들을 이용하여 정적인 작업들을 정해진 실행순서에 따라 스케줄하고, 동적인 작업이 도착하면 EDF(Earliest Deadline First) 스케줄링 방식으로 처리한다. 그러나 선행스케줄링에서는 자원공유문제를 고려하지 않고 실행시간을 부여하였으므로 여러 정적인 작업들이 하나의 자원을 공유할 경우에 배타적인 자원접근을 보장하지 못한다. 본 논문에서는 단일처리기 환경에서 여러 정적인 작업들의 자원공유를 고려하여 자원의 배타적 사용을 보장하는 선행스케줄 생성기법을 제시한다. 이 기법은 각 작업의 자원 방출시간을 예측하고 예측시간에 근거하여 각 작업의 자원사용구간이 중복되지 않도록 실행시간을 결정한다.

1. 서 론

실시간 임베디드 시스템은 자원이 제한된 환경에서 엄격한 시간제약을 만족해야 하는 특징을 가지고 있으며, 단일처리기 환경에서 비교적 적은 수의 작업들을 스케줄 하는 것이 일반적이다. 그러나 최근에는 자동화 로봇, 자동차의 제어시스템, 멀티미디어 서버 등의 실시간 응용이 다양해짐에 따라 작업부하가 더욱 복잡하고 커지면서 좀 더 효율적인 스케줄링 방식을 요구하고 있다. 스케줄링 방식은 일반적으로 오프라인과 온라인 스케줄링 방식으로 구분한다. 오프라인 방식은 작업들을 수행하기 전에 시간제약, 선행제약, 상호배제 등을 만족하는 스케줄을 확정하므로 스케줄러의 구현이 간단하고 시스템 부하가 적다는 장점이 있으나 작업부하가 가변적인 동적인 작업을 효과적으로 처리하지 못하는 단점을 가지고 있다. 이에 반해 온라인 방식은 작업들을 수행하면서 스케줄을 결정하므로 작업부하 변화를 유연하게 수용 할 수 있는 장점은 있지만 최적의 스케줄을 생성하기 어렵고 처리기 이용률이 낮다는 단점이 있다. 그리하여 온라인 스케줄링 방식과 오프라인 스케줄링 방식의 장점을 유지하면서 단점을 보완하기 위한 방법으로 "Weirong Wang, Aloysius K." 등이 선행스케줄링 방식을 제안하였다[1,2]. 이 방식은 오프라인 컴포넌트와 온라인 컴포넌트로 구성되었으며 오프라인 컴포넌트에서는 정적인 작업들에 대한 선행스케줄 정보를 생성하고, 온라인 컴포넌트에서는 오프라인에서 생성한 선행스케줄 정보를 이용하여 정적인작업과 동적인 작업을 유연하게스케줄 한다. 선행스케줄 정보는 크게 3단계를 거쳐 생성된다. 첫 번째 단계에서는 비주기적인 동적인 작업들을 고려하면서, 주기적인 정적인 작업들의 상호관계에 근거하여 정적인 작업들을 여러 부분작업으로 분할하고 그들의 실행순서를 결정한다. 두 번째 단계에서는 각 부분작업들에게 준비시간과 마감시간을 부여하고, 마지막 단계에서는 각 부분작업들의 실행시간을 결정한다. 온라인에서 각 부분작업들은 결정된 실행순서에 따라 실행하며 동적인 작업이 도착하면 EDF 방식으로 스케줄 한다. 따라서 이 방식은 정적인 작업에 대해서 실시간 특성과 같은 시간제약을 보장하면서 동적인 작업을 효과적으로 스케줄 할 수 있으므로 정적인 작업들과 동적인 작업들을 함께 스케줄 함에 있어서 효과적인 방식이다. 그러나 정적인 작업들의 배타적인 자원 공유를 고려하지 않았으므로 상호배제 문제가 발생할 수 있다.

이 논문은 2005년 정부(교육인적자원부)의 재원으로 한국학술진흥재단의 지원을 받아 수행된 연구임(KRF-2005-041-D00636)

지금까지 자원공유에 있어서 상호배제, 우선순위 역전 (priority inversion), 교착상태(deadlock) 등 문제를 해결하기 위해 priority inheritance, priority ceiling protocol, stack based resource sharing policy 등 방법이 제안되었다[3,4]. 이들은 모두 우선순위-기준(priority-driven)스케줄링 방식을 기반으로 하는 온라인 스케줄링 방식에 적용될 수 있지만, 선행스케줄링 방식에 적용하기에는 적합하지 않다. 왜냐하면 선행스케줄링 방식에서 각 부분작업들은 할당된 실행시간만큼 반드시 실행해야 되는데, 임의의 한 부분작업이 할당된 시간만큼 실행하고 나서 점유한 자원을 방출하지 않으면 순서적으로 뒤이어서 자원을 요청하는 작업들은 모두 블록(block)되고 제한된 시간내에 할당된 실행시간만큼 실행하지 못하여 마감시간을 만족하지 못하게 된다.

본 논문에서는 단일처리기 환경에서 여러 정적인 작업들이 하나의 자원을 공유하면서 동적인 작업들을 효과적으로 스케줄 하는 방법을 제시한다. 논문에서 제시하는 방법은 기존 선행스케줄링 방식을 바탕으로 하며 GC(Generating Constraints) 함수와 제약조건 생성기가 추가된다. GC 함수는 자원을 공유하는 작업들의 자원방출시간을 예측하고, 제약조건 생성기는 자원방출예측시간에 근거하여 각 작업에 대해 자원사용제약조건을 생성한다.

본 논문의 구성은 다음과 같다. 2장에서는 기존 선행스케줄링 방식에 대해서 살펴보고, 그리고 기존 방식에서 여러 작업들이 자원을 공유할 경우에 발생하는 자원 상호배제 문제를 제기하고, 3장에서는 자원을 상호 배타적으로 사용하면서 효율적으로 스케줄 할 수 있는 선행스케줄 생성기법을 제시한다. 그리고 4장에서 결론 및 향후 연구방향에 대해 기술한다.

2. 선행스케줄링의 개관

2.1 시스템 모델 및 정의

본 논문에서는 실시간 시스템의 모든 작업들이 단일처리기 환경에서 스케줄 되고 모든 작업들은 선점 가능하다고 가정하며, 여러 정적인 작업들이 하나의 자원을 공유한다고 가정한다. 그리고 전체 시스템 부하는 정적인 작업과 동적인 작업들의 집합으로 간주한다.

온라인 실행은 일반적으로 일정한 길이의 무한개의 하이퍼 간격(hyper interval)으로 분해되는데 이를 하이퍼 주기(hyper period)라고 하며 P 로 표시한다. 이는 정적인 작업과 동적인 작업을 포함한 모든 작업 실행주기의 최소공배수이다. 또한 논문에서는 설명의 편리를 위하여 정적인 작업들의 집합은 J_P 로, 동적인 작업들의 집합은 J_S 로, 자원을 사용하는 작업들의 집합은 $J_R(J_R \subseteq J_P)$로 표시한다. 그리고 정적인 작업 j_i는 (r_i, d_i, e_i)로 정의되며, r_i는 준비시간을, d_i는 마감시간을, e_i는 실행시간을 의미한다. 따라서 만약 정적인 작업 j_i가 n^{th} 하이퍼 주기에 도착한다면 이 작업은 반드시 시간 $(nP+r_i, nP+d_i)$ 동안에 스케줄 되어야 한다. 그리고 동적인 작업 j_s는 (p_s, d_s, e_s)로 정의되며, p_s는 작업 j_s가 한번 실행되고 나서 또다시 실행되기까지의 최소 시간간격을 의미한다. 자원을 사용하는 작업 j_r은 (l_r, u_r)로 정의되며, l_r은 j_r의 실제 실행시작 시간으로부터 자원을 요청하는 시간까지의 시간간격을 의미하고, u_r는 자원을 사용하는 동안의 자원 점유시간을 의미한다.

예제 1. 아래 예제에서 정적인 작업 집합 J_P와 동적인 작업 집합 J_S에 있는 모든 작업들의 주기는 225 이고, 따라서 하이퍼 주기 P는 225이다.

$$J_P = \{j_0(60,160,25), j_1(45,70,5), j_2(40,120,30), j_3(100,140,15), j_4(0,225,120), j_5(190,225,5)\}$$
$$J_S = \{j_s(225,50,25)\}$$
$$J_R = \{j_0(0,10), j_2(5,5), j_4(30,30)\}$$

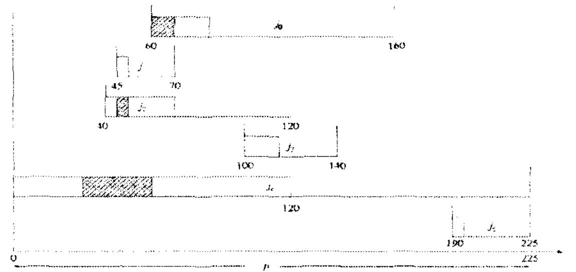

그림 1. 정적인 작업집합

그림 1은 정적인 작업들을 보여주고 있다. 수직선은 정적 작업들의 준비시간과 마감시간을 각각 지시하고 그 사이의 박스는 해당 작업 실행시간을 의미하고, 음영으로 채워진 부분은 자원을 공유하는 작업들이 자원 점유시간을 의미한다.

2.2 오프라인 컴포넌트

오프라인에서는 각 부분작업들의 실행순서와 실행시간이 결정된다. 각 부분작업들의 실행순서는 정적인 작업들의 상호관계에 근거하여 결정된다. 즉, 작업 j_a 와 j_b가 있고 $(r_a < r_b) \wedge (d_a > d_b)$이면 j_a는 j_b을 포함하며 $j_a > j_b$로 표시하고, $(r_a < r_b) \wedge (d_a \le d_b)$이면 j_a는 j_b 보다 전행되며 $j_a \rightarrow j_b$로 표시한다. 만약 작업 j_a가 임의의 한 정적인 작업이라도 포함하고 있으면 작업 j_a는 여러 개의 부분작업으로 분할되며 작업 j_a를 non-top작업이라고 하고 이 작업들의 집합을 J_N로 표시한다. 만약 작업 j_b가 임의의 한 정적 작업이라도 포함하지 않으면 작업 j_b는 여러 개의 부분작업으로 분할되지 않으며 top작업이라고 하고 이 작업들의 집합을 J_T로 표시한다. 시간적으로 연속되는 두 개의 top작업 사이를 갭(gap)이라고 하며 G로 표시한다. 갭은 갭의 시작을 나타내는 시작경계작업과 갭의 끝을 나타내는 끝경계작업으로 표시되며 이 사이는 부분작업들로 구성된다. 만약 작업 $j_n(j_n \in J_N)$이 작업 $j_t(j_t \in J_T)$을 포함하고 있으면 j_n은 j_t을 경계작업으로 하는 갭에 분할되어 부분 실행된다. 각 부분작업은 j_n^i로 표시하며 이는 작업 j_n의 i^{th} 부분작업임을 의미한다. 따라서 가장 먼저 실행되는 부분작업을 시작부분작업, 제일 마지막으로 실행되는 부분작업을 마지막부분작업이라고 정의한다. 만약 작업 $j_a(j_a \in J_N)$와 $j_b(j_b \in J_N)$가 각각 여러 개의 부분작업들로 분할되고, 작업 j_b 시작부분작업의 실행순서가 작업 j_a 마지막부분작업의 실행순서보다 늦다면 작업 j_a와 j_b는 상호배제문제가 발생하지 않으며 $j_a \Phi j_b$로 표시한다. 그리고 각각의 작업들을 분해하고 나서 같은 갭에 존재하는 부분작업들의 실행순서 결정은 다음과 같다.

- G_i에서 $j_a \rightarrow j_b$ 이면 G_i에 존재하는 j_a의 부분작업은 j_b의 부분작업보다 항상 빠른 실행순서를 가진다.

- G_i에 $j_a > j_b$이면 세 가지 경우로 생각할 수 있다. 즉, (1) 작업 j_a가 G_i의 끝경계작업을 포함하고 작업 j_b는 G_i의 끝경계작업을 포함하지 않는 경우, (2) j_a와 j_b가 모두 G_i의 끝경계작업을 포함하는 경우, (3) j_a와 j_b가 모두 G_i의 끝경계작업을 포함하지 않는 경우이다. 세 경우에 있어서 실행순서 결정은 (1)인 경우에는 G_i에서 j_b의 부분작업은 j_a의 부분작업보다 빠른 실행순서를 가진다. (2)와 (3)인 경우에는 G_i에서 j_a의 부분작업은 j_b의 부분작업보다 빠른 실행순서를 가진다.

다음으로 각 부분작업들의 준비시간과 마감시간은 아래와 같이 부여한다. G_i에 존재하는 작업 j_a의 부분작업에 대해 준비시간과 마감시간을 부여할 때 만약 r_a가 G_i의 시작경계작업의 준비시간보다 작거나 같으면 시작경계작업의 준비시간을 준비시간으로 부여한다. 만약 r_a가 G_i의 시작경계작업의 준비시간보다 크면 r_a를 부분작업의 준비시간으로 부여한다. 만약 d_a가 G_i의 끝경계작업의 마감시간보다 작거나 같으면 d_a를 마감시간으로 부여한다. 만약 d_a가 G_i의 끝경계작업의 마감시간보다 크면 끝경계작업의 마감시간을 부분작업의 마감시간으로 부여한다.

마지막으로 각 부분작업들의 실행시간을 결정한다. 각 부분작업들의 실행시간은 non-negative constraints, sufficiency constraints, slack-reserving constraints 제약 조건을 LP에 적용하여 구해진다. Non-negative constraints는 각 부분작업들의 실행시간은 0보다 크거나 같아야 되고, sufficiency constraints는 한 작업이 n 개의 부분작업으로 분할되었다고 가정할 때 n개 부분작업들의 실행시간의 합은 해당 작업의 실행시간과 같아야 되고, slack-reserving constraints는 비주기적으로 도착하는 동적인 작업을 처리할 시간을 확보하고 나서 주기적으로 도착하는 정적인 작업을 처리하는데 주어진 시간을 의미한다.

예제 1에서 정적인 작업들의 상호관계에 근거하여 non-top작업 집합은 $J_N = \{j_0, j_2, j_4\}$이고, top작업 집합은 $J_T = \{j_1, j_3, j_5\}$이다. 즉, 작업 j_0은 작업 j_3을 포함하고 있으므로 작업 j_3을 끝경계작업으로 하는 G_1에 j_0^0과 작업 j_3을 시작경계작업으로 하는 G_2에 j_0^1로, j_2는 작업 j_1를 포함하고 있으므로 작업 j_1을 끝경계작업으로 하는 G_0에 j_2^0과 작업 j_1을 시작경계작업으로 하는 G_1에 j_2^1로, j_4는 작업 j_1과 j_3을 모두 포함하고 있으므로 같은 방법으로 (j_4^0, j_4^1, j_4^2)로 분할된다. 그리고 작업 j_2와 j_0는 상호배제 관계이므로 작업 j_0의 부분작업은 반드시 작업 j_2가 실행을 끝마친 후에 실행을 시작할 수 있으므로 $j_2 \Phi j_0$이다. 따라서 오프라인에서 생성한 선행스케줄 정보 F는 다음과 같다.

$$F = \{j_4^0(0,70,40), j_2^0(40,70,5), j_1(45,70,5), j_4^1(45,140,10), j_2^1(45,120,25), j_0^0(60,140,15), j_3(100,140,15), j_0^1(100,225,10), j_4^2(100,225,70), j_5(190,225,5)\}$$

2.3 온라인 컴포넌트

온라인 컴포넌트는 오프라인에서 생성한 선행스케줄 정보를 이용하여 각 부분작업들을 정해진 실행순서에 따라 스케줄 하며 동적인 작업이 도착하면 EDF 방식으로, 즉, 마감시간이 빠른 작업을 먼저 실행한다. 예제 1에서 시간 30에 동적인 작업이 도착하면 온라인 스케줄러는 그림 2와 같이 스케줄 한다.

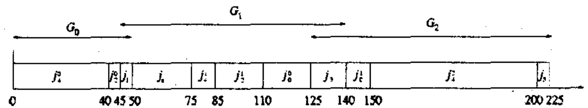

그림 2. 선행스케줄 정보 F

2.4 상호배제 문제

앞에서 설명한 바와 같이 선행스케줄링 방식은 작업들을 부분작업으로 분할하고, 그들의 실행순서 결정, 도착시간과 마감시간을 부여, 실행시간 결정하는 3단계 처리과정을 거친다. 예제 1에서 이렇게 생성된 선행스케줄 정보 F에 따라 스케줄 할 때 만약 여러 작업들이 하나의 자원을 공유할 경우에 그림 3과 같이 자원 상호배제문제가 발생한다.

그림 3에서 (j_0와 j_4), (j_2와 j_4)는 각각 자원 공유에 있어서 상호배제 문제가 발생하였다. 즉, 부분작업 j_4^0이 실행을 끝마치고 나서 자원을 방출하지 않았기 때문에 부분작업 j_2^1가 자원 사용을 요청하면 블록 된다. 뿐만 아니라 뒤이어서 자원을 요청하는 부분작업 j_0^0도 자원을 접근하지 못하고 블록 되었다. 이는 각 부분작업들의 실행시간 할당에 있어서 자원 공유를 고려하지 않고 실행시간을 부적절하게 할당했기 때문이다.

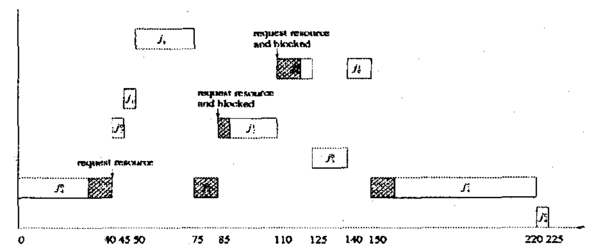

그림 3. 상호배제 문제 발생

3. 자원 공유 선행스케줄링

선행스케줄에서 배타적으로 자원을 접근하기 위해서는 사용되고 있

는 자원에 대해 접근을 요청하지 말아야 한다. 따라서 자원을 상호 배타적으로 사용하기 위해서는 자원을 점유한 부분작업은 자원 사용을 끝마칠 때까지 계속 실행해야 되거나 아니면 자원 사용을 끝마칠 때까지 다른 작업들은 자원접근요청을 못하도록 해야 한다.

본 논문에서는 이상의 조건에 근거하여 자원사용제약조건을 생성하고 생성된 제약조건을 LP에 추가 적용하여 각 부분작업들의 적절한 실행시간을 결정한다. 자원사용제약조건은 자원을 공유하는 작업들의 자원 방출시간을 예측하고 예측된 시간에 근거하여 생성된다. 자원방출예측시간은 v로 표시하고 v_i는 작업 j_i의 자원 방출 예측시간을 의미하고 이는 GC 함수에서 계산되며 제약조건 생성기는 예측된 자원 방출시간과 아래의 제약조건에 근거하여 자원사용제약조건을 생성한다.

(1) 자원을 점유하고 있는 작업이 자원을 방출할 때까지 다른 작업은 자원접근요청을 하지 말아야 한다.

(2) 만약 작업 j_a가 j_b에 포함되면 작업 j_a는 작업 j_b를 선점 가능하다.

3.1 GC 함수

GC 함수는 자원을 공유하는 작업들의 자원접근순서를 결정하고 자원방출시간을 예측한다. 자원방출예측시간을 계산하기 위해서는 자원접근순서가 먼저 정해져야한다 자원접근순서는 자원을 공유하는 작업의 $r+l$으로 결정하며 작은 값을 가지는 작업에 빠른 접근순서를 부여하고, 같은 값을 가질 경우에는 마감시간이 빠른 작업에 빠른 자원접근순서를 부여한다. 그리고 자원접근순서에 따라 순차적으로 각 작업의 자원방출예측시간을 계산한다. 예를 들어 자원접근 대기 열 $Q=(j_i,\cdots,j_k,\cdots,j_n)$가 있고 j_i는 자원을 제일 먼저 접근하는 작업, j_n은 자원을 제일 마지막에 접근하는 작업, j_k는 이들 사이의 작업이며 자원방출예측시간은 순차적으로 계산된다. 그러나 $j_{k-1} > j_k$이고 j_{k-1}가 자원을 접근하기 전에 작업 j_k가 도착하면 제약조건 (2)에 근거하여 작업 j_k는 j_{k-1}를 선점하므로 작업 j_{k-1}는 j_k가 자원을 방출할 때까지 자원 접근을 요청하지 말아야한다.

자원방출예측시간 계산은 다음과 같다. 자원을 제일 처음으로 접근하는 작업은 블록 되지 않고 제일 높은 우선순위를 가지므로 작업 j_i의 자원방출예측시간 계산은 식 (1)에 근거하고, $j_k(j_k \in J_N)$의 자원방출예측시간 계산에 있어서는 j_{k-1}가 자원 사용을 끝마치고 나서 j_k가 도착하는 경우와 j_{k-1}가 자원 사용을 끝마치기 전에 j_k가 도착하는 경우로 생각해볼 수 있다. 이때 j_k의 자원방출예측시간 계산은 전자의 경우 식(2)에 근거하고 후자의 경우 식(3)에 근거한다. 그러나 작업 $j_{k-1}\Phi j_k$이고 Q에서 작업 j_{k-1}의 자원접근순서가 j_k보다 빠를 경우에는 작업 j_k는 작업 j_{k-1}가 실행을 끝낸 후에야 실행을 시작할 수 있으므로 j_k의 자원 방출 예측시간 계산 시 $l_{k-1}+u_{k-1}$대신 실행시간 e_{k-1}을 사용하여 식(4)에 근거한다.

$$v_i = r_i + l_i + u_i \quad (1)$$

$$v_{k-1} < r_k \text{이면 } v_k = r_k + (l_k + u_k) \quad (2)$$

$$v_{k-1} \geq r_k \text{이면 } v_k = v_{k-1} + (l_k + u_k) \quad (3)$$

$$j_{k-1}\Phi j_k \text{이면 } v_k = v_{k-2} + e_{k-1} + (l_k + u_k) \quad (4)$$

예제 2. 위의 알고리즘에 따라 예제 1에서 자원공유 작업들의 자원접근순서는 $Q=(j_4, j_2, j_0)$이고, 작업 j_4는 자원을 제일 먼저 접근하는 작업이므로 식(1)에 근거하고, 작업 j_2의 자원방출예측시간 계산은 $v_4 > r_2$이므로 식(3)에 근거하고, 작업 j_0의 자원방출예측시간 계산은 $v_2 > r_0$ $v_2 > r_0$이므로 식(3)에 근거하여 계산한 결과는 다음과 같다.

$$v_4 = 65;\ v_2 = 75;\ v_0 = 85;$$

3.2 제약조건 생성

자원사용제약조건은 자원방출예측시간에 근거하여 제약조건 생성기에서 생성한다. 자원사용제약조건을 생성하기 위해서는 자원을 공유하는 작업들이 어느 갭에서 자원 사용을 끝낼 수 있는지를 알아야 한다. 즉, 작업 $j_r(j_r \in J_R)$의 자원방출예측시간 v_r을 계산하고 아래 식(5)를 만족하는 $j_t(j_t \in J_T)$을 찾는다.

$$v_r \leq d_t - \sum_{j_t \in J_T}^{j_t} e_t - slack(j_t) \quad (5)$$

설명의 편리를 위해 식(5)의 $d_t - \sum_{j_t \in J_T}^{j_t} e_t - slack(j_t)$를 T_t로 표기한다. 여기서 $\sum_{j_t \in J_T}^{j_t} e_t$는 선행스케줄 정보 F에서 j_t까지 존재하는 *top* 작업들의 실행시간의 합이고, $slack(j_t)$는 작업 j_t가 실행되기까지 비주기적으로 도착하는 동적인 작업을 처리하기 위해 확보된 시간이다. 만약 자원을 작업 j_r의 여러 부분작업에서 분할 사용한다면 이 사이에 스케줄 되는 다른 부분작업들은 자원접근을 요청하지 말아야 하는 제약조건을 생성한다. 작업 j_r의 $v_r \leq T_t$이면 j_t를 끝경계작업으로 하는 G_i안에서 자원 사용을 끝낼 수 있으므로 선행스케줄 정보 F에서 작업 j_t까지 존재하는 작업 j_r 부분작업들의 실행시간의 합은 $l_r + u_r$ 보다 크거나 같아야 되고, 이들 사이에 스케줄 되는 자원공유 작업들의 부분작업의 실행시간 합은 해당 작업의 l보다 작아야 한다. 그리고 자원을 공유하는 작업이 top 작업일 경우, 즉, $(j_r \in J_T) \wedge (j_r \in J_R)$이면 j_r이 여러 개의 부분작업으로 분할되지 않으므로 (a)자원 접근 대기열 Q에서 작업 j_r 전에 정렬된 자원을 공유하는 모든 작업들은 반드시 자원사용을 끝내거나, (b)아니면 자원접근을 요청하지 말아야 한다. 이에 근거하여 (a)인 경우 제약조건 (1)을 생성하고 (b)인 경우 제약조건 (2)를 생성한다.

(1). j_r 앞에 존재하는 자원을 공유하는 각 작업들의 부분작업의 실행시간 합은 해당 작업의 $l+u$보다 크거나 같아야 한다.

(2). j_r 앞에 존재하는 자원을 공유하는 각 작업들의 부분작업의 실행시간 합은 해당 작업의 l보다 작아야 한다.

예제 3. 작업 j_4는 $r_4 < T_3$이므로 j_3을 끝경계작업으로 하는 G_1에서 자원 사용을 끝낼 수 있다. 그러므로 G_1까지 분할된 작업 j_4의 부분작업들 실행시간 합은 $e_4^0+e_4^1 \geq l_4+u_4$되며, 이 사이에 스케줄 되는 작업 j_2의 부분작업 $e_2^0 < l_2$이여야 한다. 작업 j_2도 $r_2 < T_3$이므로 작업 j_3을 끝경계작업으로 하는 G_1에서 자원 사용을 끝낼 수 있다. 그러므로 작업 j_2의 부분작업 합은 $e_2^0+e_2^1 \geq l_2+u_2$여야 한다. 그리고 $j_0\Phi j_2$를 고려하여 생성한 자원사용제약조건들은 식(6)과 같다.

$$e_4^0+e_4^1 \geq 30;\ e_2^0 < 5;\ e_2^0+e_2^1 \geq 10: \quad (6)$$

식(6)의 자원사용제약조건을 고려하여 생성한 선행스케줄 정보 F'는 다음과 같다.

$$F' = \{j_4^0(0,70,45), j_2^0(40,70,0), j_1(45,70,5), j_4^1(45,140,15), j_2^1(45,120,30), j_0^0(60,140,5), j_3(100,140,15), j_0^1(100,225,20), j_4^2(100,225,60), j_3(190,225,5)\}$$

그림 4는 동적인 작업 j_x가 시간 30에 도착할 경우에 자원을 배타적으로 사용하면서 모든 작업들이 효과적으로 스케줄 되고 있음을 보여주고 있다.

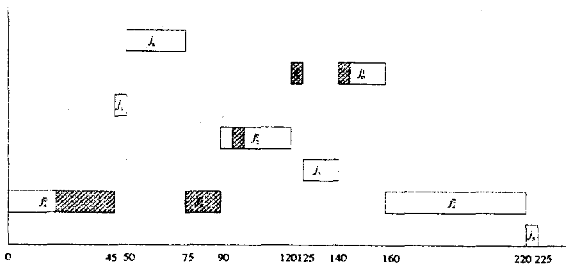

그림 4. 자원 공유 선행스케줄

4. 결론 및 향후 연구

선행스케줄링에서 각 부분작업들은 오프라인 컴포넌트에서 할당된 시간만큼만 실행된다. 그러나 자원을 사용중인 임의의 한 부분작업이 할당된 실행시간만큼 실행되고 나서 자원을 방출하지 않으면 순서적으로 뒤이어서 자원접근요청을 하는 부분작업들은 블록 되어 해당 부분작업의 마감시간까지 할당된 실행시간만큼 실행을 마치지 못하게 된다. 따라서 자원을 공유하는 작업들은 정상적으로 스케줄 되지 못하고 시간제약을 만족하지 못하게 된다. 본 논문에서는 선행스케줄링 방식에서 여러 정적인 작업들이 하나의 자원을 공유하면서 비주기적으로 도착하는 동적인 작업들을 효율적으로 스케줄 할 수 있는 선행스케줄 생성기법을 제시하였다. 제시한 기법에서는 기존 선행스케줄링 방식에 따라 작업들을 여러 부분작업으로 분할하고 각 부분작업에 준비시간과 마감시간을 부여한다. 그리고 GC 함수와 제약조건 생성기를 통해 자원 사용 제약조건을 생성하고 생성된 제약조건을 LP에 추가 적용하여 적절한 실행시간을 할당하였다. 그러나 본 논문에서 제시한 방법은 자원을 공유하는 정적인 작업들만 고려하였고 비주기적으로 도착하는 동적인작업의 자원공유에 대해서는 고려하지 않았다. 따라서 향후에는 이에 대한 연구가 필요하다.

참 고 문 헌

[1] Weirong Wang, Aloysius K. Mok, Gerhard Fohler, "Pre-Scheduling: Integrating Offline and Online Scheduling Techniques",The Conference on Embedded Software (EMSOFT), pp.356-372, 2003.

[2] Weirong Wang, Aloysius K.Mok, and Gerhard Fohler. "Pre-Scheduling", Real-Time Systems, Vol. 30, Numbers 1-2, pp 83-103, 2005.

[3] Lui Sha, Ragunathan Rajkumar, John P. Lehoczky, "Priority Inheritance Protocols: An Approach to Real-Time synchronization", IEEE Transactions on Computers. Vol. 39, No. 9, 1990.

[4] T. P. Baker. "A Stack-based Resource Allocation Policy for Realtime Processes", In Proceedings of the IEEE Real-Time Systems Symposium, pp.191-200, 1990.