실시간 분산처리 시스템에서의 고신뢰 구조연구

연구수행기관 : 한국과학기술원

한국전자통신연구소

제 출 문

한국전자통신연구소 귀하

“실시간 분산처리 시스템에서의 고신뢰 구조연구”에 관한 위탁연구의 1차년도 최종보고서를 별첨과 같이 제출합니다.

1994년 9월 30일

수행 연구 기관 : 한국과학기술원

연 구 책 임 자 : 이 홍 규 (인)

수행연구기관장 : 심 상 철(직인)

요 약 문

1. 제 목

실시간 분산처리 시스템에서의 고신뢰 구조 연구

2. 연구의 목적 및 중요성

1) 연구의 목적

본 연구의 목적은 엄격한 실시간 시스템을 위한 운영체제의 핵심 부분인 스케쥴링 알고리즘을 조사 분석하고, 차세대 전전자 교환기에 적합한 실시간 스케쥴링 알고리즘을 개발하는 것이다.

2) 연구의 중요성

과거, 실시간 처리는 군사 무기 시스템, 원자력 발전소 제어등과 같은 특징 응용 분야에만 사용되었다. 그러나 하드웨어의 기술 발전과 이에 의한 가격 저하 및 각종 응용 기술의 발전, 갈수록 복잡해지고 질적 향상을 기대하는 사용자의 요구 증대 등으로 인하여 멀티미디어 및 각종 통신 분야 에서도 실시간 응용기술이 필요하게 되었다.
실시간 스케쥴링 알고리즘에 관한 연구는 이러한 실시간 처리 기술의 핵심 분야로서 갈수록 복잡해지고 규모가 커지는 실시간 시스템의 설계 및 운영에 있어서 없어서는 안 될 스케쥴링 가능성을 통한 예측성과 수행 순위 결정을 제공한다.

3. 연구의 내용 및 범위

1) 실시간 스케쥴링 알고리즘에 대한 기존의 연구 조사 및 분석
2) 고장 허용성 실시간 스케쥴링 알고리즘에 대한 기존의 연구 조사 및 분석

3) 차세대 전전자 교환기의 실시간 특성 추출 및 분석
4) 차세대 저 전자 교환기 운영체제를 위한 고장 허용성 스케쥴링 알고리즘의 설계 및 분석

4. 연구 결과

1) 실시간 스케쥴링 알고리즘에 대한 기존의 연구 조사 및 분석
2) 차세대 교환기의 실시간 처리가 요구되는 특성 추출 및 분석
3) 스케쥴링 알고리즘의 구현과 분석

5. 활용에 대한 건의

차세대 전전자 교환기의 구현에 이용될 수 있다. 개발될 스케쥴링 알고리즘과 실시간 운영체제 구조를 사용하여 시스템을 구축에 이용하면, 시스템의 시간적으로 정확한 동작을 보장할 수 있게 된다. 따라서 이러한 시스템들의 개발 시간을 크게 단축시킬 수 있으며, 신뢰성 있는 고품질의 시스템 구축이 가능해진다. 또한, 각종 실시간 제어 시스템들, 즉 소형 인공위성, 엔진 관리 시스템(EMS), robot system 등에 개발된 관련기술들을 활용할 수 있을 것이다.

6. 기대효과

1) 실시간 스케쥴링 알고리즘 설계기술 확보
2) 실시간 스케쥴링 알고리즘 Simulation 환경 확보
3) 차세대 교환기에 필요한 실시간 스케쥴링 알고리즘 확보
4) 차세대 교환기에 필요한 실시간 운영체계 설계기술 확보
5) 실시간 운영체제 개발의 핵심 기술 확보

목 차

제 1 장 서 론
1.1 연구의 필요성
1.2 연구의 개요
1.3 연구의 목표

제 2 장 기존의 실시간 스케쥴링 알고리즘
2.1 서 론
2.2 정적 스케쥴링 알고리즘(Static Scheduling Algorithm)
2.2.1 중앙 집중 시스템에서의 정적 스케쥴링 알고리즘
2.2.2 분산시스템에서의 정적 스케쥴링 알고리즘
2.3 동적 실시간 스케쥴링 알고리즘
2.3.1 중앙 집중 시스템에서의 동적 실시간 스케쥴링 알고리즘
2.3.2 분산 시스템에서의 동적 실시간 스케쥴링 알고리즘
2.4 결함 허용성 실시간 스케쥴링 알고리즘 (Fault Tolerant Real Time Scheduling Algorithm)
2.4.1 단일 처리기 결함 허용성 스케쥴링 알고리즘
2.4.2 다중 처리기 결함 허용성 스케쥴링
2.5 결 론

제 3 장 교환기 운영체제에서의 실시간 처리 요구분석
3.1 교환기 운영체제의 구조 분석
3.2 차세대 전 전자교환기 운영체제의 요구사항
3.3 차세대 전 전자교환기를 위한 스케쥴링 알고리즘의 요구사항

제 4 장 제안된 실시간 스케쥴링 알고리즘
4.1 서 론
4.2 기존의 스케쥴링 알고리즘
4.2.1 폴링 서버(polling server)
4.2.2 슬랙 스틸링(slack stealing)
4.2.3 문제점 분석 및 문제 정의
4.3 비주기 태스크의 응답 시간 개선을 위한 이중 우선순위 알고리즘
4.3.1 혼합 태스크 스케쥴링의 환경 및 가정
4.3.2 기본적 발상
4.3.3 이중 우선순위 알고리즘
4.3.4 이중 우선순위 할당의 정당성
4.3.5 슬랙 스틸링의 개선
4.4 실험 및 평가
4.4.1 주기의 형태에 의한 영향 실험
4.4.2 고유한 우선순위를 가지는 비주기 태스크의 실험
4.5 결 론

제 5 장 결론 및 앞으로의 연구 방향
5.1 결론
5.2 앞으로의 연구 방향

그림 목차

1.1 실시간 스케쥴링 알고리즘의 분류

2.1 Flow Network의 예
2.2 스케쥴 가능 조건
2.3 next-fit 알고리즘
2.4 first-fit 알고리즘
2.5 L-C 평면
2.6 Myopic 알고리즘의 휴리스틱 함수
2.7 분산 시스템에서의 실시간 스케쥴러의 구성
2.8 Focused Addressing
2.9 단순 주기
2.10 알고리즘 1
2.11 단순 주기 태스크의 예
2.12 알고리즘 1의 스케쥴링 과정
2.13 스케쥴링의 사례
2.14 알고리즘 2
2.15 알고리즘 3에서 구성된 스케쥴 트리
2.16 조건 2의 예
2.17 알고리즘 Q

3.1 교환기 시스템의 기본적인 구조
3.2 시스템의 구조

4.1 Hole
4.2 PE알고리즘에서 발생하는 최악의 대기시간

4.3 Lehoczky의 정리
4.4 슬랙 스틸링 알고리즘에서 가용 슬랙 시간의 계산
4.5 임의 시점에서의 슬랙 시간의 결정
4.6 슬랙 스틸링의 단점
4.7 Lehoczky 분석의 단점
4.8 혼합 스케쥴러의 구조
4.9 낮은 우선순위 태스크에 의한 상위 우선순위 태스크의 슬랙 조정
4.10 RM 스케쥴링에 의한 결과
4.11 태스크의 분할
4.12 디스펫쳐의 비교
4.13 우선순위의 할당
4.14 보조 정리 1의 예
4.15 개선된 슬랙 스틸링의 수행
4.16 주기 차이가 작은 태스크 집합
4.17 주기 차이가 큰 태스크 집합
4.18 INS에서의 실험결과

표 목 차

4.1 주기 태스크의 예
4.2 주기 태스크의 예
4.3 주기 태스크의 예
4.4 이중 우선순위 할당의 결과
4.5 주기의 차이가 큰 주기 태스크 집합
4.6 주기의 차이가 작은 태스크의 집합
4.7 INS주기 태스크

제 1 장 서론

1.1 연구의 필요성

실시간 시스템은 계산의 결과가 논리적으로 정확해야 할 뿐 아니라, 결과가 생성되는 시간적 제약 조건(timing constraints)또한 정확히 만족해야 하는 시스템이다. 따라서, 태스크들(tasks)이 시간적 제약 조건을 지니고 있는 시스템인 전전자 교환기는 실시간 시스템의 좋은 응용 예이다. 만일, 이러한 시스템에서 시간적 조건에 만족하지 못하는 작동을 하였을 경우, 실시간 시스템에 의해 운용되는 전체 시스템에 심각한 장애를 일으키며 인간에게까지 그 영향을 미친다.

또한, 이러한 실시간성에 따른 동적인 고장과 더불어, 하드웨어 고장 또는 소프트웨어 장애의 발생과 같은 정적인 고장에도 불구하고, 고장이 전체 시스템의 동작에 미치는 영향을 최소화하는 신뢰성 문제는 매우 중요한 문제이다.

이러한 실시간 시스템의 시간에 대한 정확한 동작을 위해서는 운영 체제의 시간에 대한 예측성이 필수적이다. 실시간 운영체제의 시간에 대한 명확성의 핵심 부분이 이 운영 체제가 수행을 관리하는 실시간 시스템의 동작을 실제로 행하는 작업들의 스케쥴링 알고리즘이다.

실시간 스케쥴링 알고리즘에서의 문제점을 살펴보면 다음과 같다. 비선점 스케쥴링은 대개 NP-hard 이므로, 적절한 휴리스틱(Heuristic) 알고리즘이 개발되어야 하지만, 단일 프로세서의 경우조차 효율적인 휴리스틱 알고리즘이 알려져 있지 않다. 이와 같은 문제의 복잡성 때문에, 비선점 스케쥴링은 선점 스케쥴링에 비해 상대적으로 연구가 덜 되어 있다.

위에서 기술한 스케쥴링 문제는 태스크들이 서로 순서관계(precedence relation)가 있거나, 자원(resource)을 요구하는 태스크들이 포함된 경우는 고려하지 않은 비교적 단순한 것이다. 단일 프로세서의 경우, 순서관계를 지닌 태스크들의 스케쥴링 문제는 최근접 한계시간 우선 방법으로 해결할 수 있다. 그러나, 다중 프로세서의 경우 이 문제는 NP-hard 문제가 된다. 선점 스케쥴링 알고리즘의 경우도 문맥 전환 등의 운용상 부가비용을 무시하고 극히 제한된 환경을 가정하고 설계 되어왔다. 따라서 실시간 시스템에 적용하기 위해서는 해당 시스템의 특성분석에 기반을 두고 만들어진 알고리즘을 사용해야 할 것이다.

본 연구에서는 엄격한 실시간 시스템을 위한 운영체제의 핵심부분인 스케쥴링 알고리즘에 대해 연구하였다. 본 연구에서는 개발된 알고리즘은 차세대 전 전자 교환기를 위한 실시간 운영체계를 위한 것으로 고장 허용성 보장과 수행시간 overhead를 최소화하는 것을 목적으로 하는 알고리즘이다.

1.2 연구의 개요

실시간 시스템(real-time system)은 연성 실시간 시스템(soft real-time system)과 엄격한 실시간 시스템(hard real-time system)으로 구분된다. 연성 실시간 시스템이란 태스크(task)에 특정한 시간제한(timing constrain ts)이 주어지지 않고, 단지 태스크의 응답시간(response time)이 가능한 한 빠를 것을 요구하는 시스템이다. 엄격한 실시간 시스템이란 시스템의 정확성이 계산의 논리적 정확성과 계산결과 도출의 시간적 정확성에 의하여 결정되는 시스템으로, 항공 제어 시스템, 무기 시스템, 공정처리 시스템, 원자로 제어 시스템과 같은 시스템을 예로 들 수 있다. 이러한 시스템은 실시간 처리요구를 유효한 시간이내에 처리하지 못하는 경우에는 심각한 결과를 초래하게 된다.

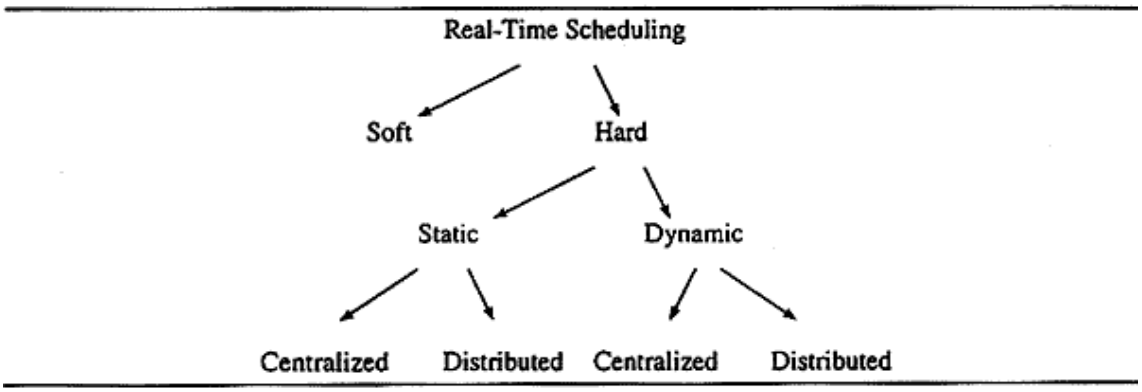


그림 1.1 : 실시간 스케쥴링 알고리즘의 분류

엄격한 실시간 시스템이 실시간 처리요구에 대응하는 태스크의 유효시간에 맞게 수행시키기 위해서는 태스크들의 수행순서를 결정하는 스케쥴링 알고리즘이 필요하다[SR88].

실시간 스케쥴링 알고리즘은 시스템의 복잡성과 태스크의 특성을 정의하는 요소들에 의하여 여러 가지로 분류된다. 즉 처리기의 개수, 자원의 제한성과 같은 조건에 따라 스케쥴링 알고리즘의 복잡성이 달라지며, 태스크가 정적인가 혹은 동적인가에 따라 알고리즘의 구성이 달라지게 된다. 그림 1.1는 이러한 기준에 의한 분류를 그림으로 나타낸 것이다[SR88].

본 연구에서는 엄격한 실시간 시스템을 위해 임 제안된 여러 알고리즘을 조사하여 그 특성을 분석하고, 이들이 가지고 있는 문제점을 분석하였고 고장 발생시, 이를 복구하기 위한 프로세스의 실시간적 처리 방법에 대하여 조사하였다. 또한, 차세대 전 전자 교환기의 운영체제가 요구하는 실시간 특성들을 연구 분석하여 이를 바탕으로 차세대 전 전자 교환기에 적합한 실시간 스케쥴링 알고리즘을 개발하고 이를 분석하였다.

1.3 연구의 목표

본 연구에서는 다음의 목표를 수행하였다.

· 실시간 스케쥴링 알고리즘에 대한 기존의 연구 조사 및 분석

· 고장 허용성 실시간 스케쥴링 알고리즘에 대한 기존의 연구 조사 및 분석

· 새로운 태스크 모델에 관한 실시간 스케쥴링 알고리즘 연구 조사 및 분석

· 차세대 전 전자 교환기의 실시간 처리가 요구되는 특성 추출 및 분석

· 차세대 전 전자 교환기의 실시간 운영체제를 위한 스케쥴링 알고리즘의 개발

본 보고서의 구성은 다음과 같다. 2장에서는 기존의 실시간 스케쥴링 알고리즘을 조사하고 분석하였다. 3장에서는 전 전자 교환기의 시스템 구조 및 특성을 알아보고 이를 통하여 스케쥴링 알고리즘의 요구사항을 분석 추출하였다. 4장에서는 3장의 내용을 바탕으로 차세대 전 전자 교환기 운영체제를 위한 스케쥴링 알고리즘을 제시한다. 5장에서는 결론과 차후 연구방향을 제시한다.

제 2 장 기존의 실시간 스케쥴링 알고리즘

2.1 서 론

70년대까지의 스케쥴링 알고리즘에 관한 연구는 실질적인 스케쥴러의 구현보다는 이론적인 바탕을 위한 연구가 근간을 이루고 있으며, 간단한 형태나 제한적인 시스템에 응용이 가능한 스케쥴링 알고리즘이 개발되었다. 80년대에 들어오면서 초기의 실시간 시스템인 단순 embeded 시스템에서 벗어나 보다 규모가 크고, 복잡한 분산실시간 시스템과 다중처리기 시스템을 위한 스케쥴링 알고리즘의 연구가 시작되었다. 또한 시스템의 거대화에 따른 고장 발생의 가능성 증대를 고려하여 고장 발생시, 이를 처리하는 프로세스의 실시간적 처리에 관한 연구도 진행되었다.

본 장에서는 기존의 실시간 스케쥴링 알고리즘과 고장허용 처리 기법을 연구 조사하고, 개발될 실시간 스케쥴링 알고리즘에서의 고장 허용을 위한 요구사항을 정리한다.

2.2 정적 스케쥴링 알고리즘 (Static Scheduling Algorithm)

본 절에서는 정적 스케쥴링 알고리즘들을 살펴본다. 정적 스케쥴링 알고리즘은 태스크의 발생시간(release time), 준비시간(ready time), 제한시간(deadline)과 최대 계산시 간(computation time)등의 정보가 미리 주어진다는 가정 하에서 off-line으로 스케쥴을 구성하는 스케쥴링 방법이다.

정적 스케쥴링 알고리즘은 수행시간 이전에 스케쥴이 구성되므로 스케쥴링의 부담(overhead)이 태스크 수행에 주는 영향이 작다는 장점이 있으나, 동적으로 발생하는 태스크를 처리할 수 없다는 단점이 있다. 정적 스케쥴링 알고리즘의 우선순위할당(priority allocation)은 고정(fixed) 우선순위할당과 동적(dynamic) 우선순위할당이 있다. 고정 우선순위 할당은 하드웨어 인터럽트와 같이 시스템 정의 시간에 우선순위가 정의되는 태스크를 효과적으로 처리할 수 있다. 반면 동적 우선순위 할당은 처리기의 사용효율이 고정 우선순위 할당보다 좋다.

본 절에서는 기존의 정적 스케쥴링 알고리즘을 중앙 집중 시스템(centralized system)에서의 스케쥴링 알고리즘과 분산시스템(distributed system)에서의 스케쥴링 알고리즘으로 분류하여 알아본다.

2.2.1 중앙 집중 시스템에서의 정적 스케쥴링 알고리즘

중앙 집중 시스템에서의 정적 스케쥴링 알고리즘은 주로 초기의 간단한 제어시스템 구성하기 위해 개발된 가장 간단한 형태의 스케쥴링 알고리즘으로서, 태스크들이 상호 연관 없이 독릭적으로 수행되는 모델과 태스크간에 수행순서가 정의되어진 모델로 구분될 수 있다.

독립적인 태스크의 스케쥴링 알고리즘
태스크들 간에 선점이 허용되는 경우, 단일 처리기(single processor)에서는 EDT(Earliest Deadline First)알고리즘이 최적(optimal)알고리즘이다[Hor74]. 최적 알고리즘이란 주어진 태스크들이 모두 제한시간이내에 스캐쥴 될 수 있으면 항상 스케쥴을 구성

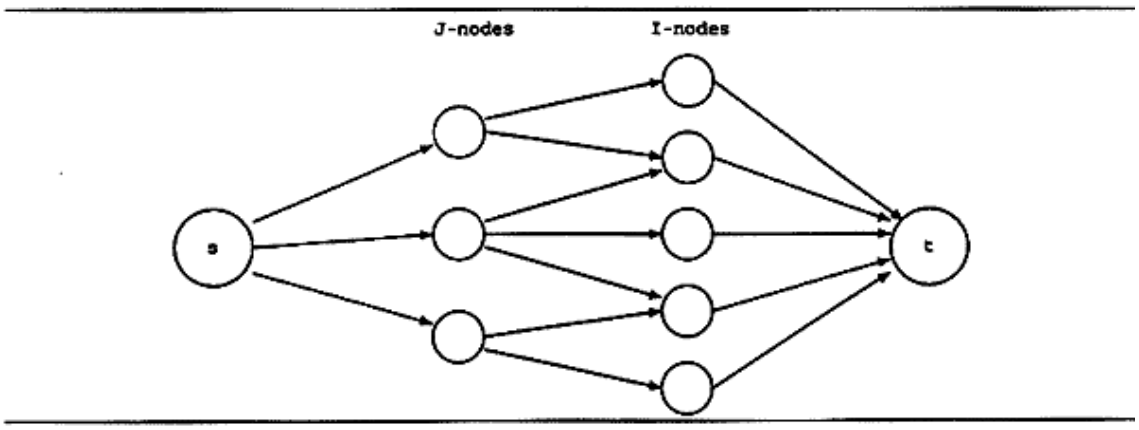


그림 2.1 : Flow Network의 예

할 수 있는 알고리즘을 의미한다. EDF 알고리즘은 가장 작은 제한시간(deadline)을 가지는 태스크에 높은 우선순위를 할당한다. EDF 알고리즘은 동적 우선순위 할당 알고리즘으로서 최적 스케쥴링 알고리즘이다.

EDF알고리즘의 변형으로는 LLF(Least Laxity First)알고리즘이 있다. [DM89]. LLF알고리즘은 가장 작은 렉시티(laxity)를 가지는 태스크에 가장 높은 우선순위를 주는 동적 우선순위 할당 알고리즘으로 EDF 알고리즘과 마찬가지로 최적 스케쥴링 알고리즘이다. 태스크 Ti의 렉시티 L_i는 다음과 같이 정의된다.

$L_i=d_i-(C_i+t)$

여기서 d_i는 태스크 T_i의 제한시간이고, C_i는 T_i의 잔여 계산시간, t는 현재시간을 나타낸다. LLF알고리즘은 EDF알고리즘보다 context switching이 많이 발생한다는 단점이 있다.

다중 처리기(multiprocessor)를 위한 스케쥴링 알고리즘으로는 Network Flow를 이용한 알고리즘이 있다. [DL78]. Network Flow 알고리즘은 그림 2.1과 같은 flow network를 이용하여 스케쥴을 구성한다. 그림 2.1에서 J-node는 각 태스크를 나타내며, P_i는 태스크 T_i의 계산시간이고, I-node는 태스크들의 준비시간과 제한시간에 의하여 구성되는 시구간을 나타낸다.

- $f((s,j)) \leq p_j$, $j = 1, 2, ..., n$;
- for all $X \subseteq B_j, f(X) \leq \rho(X), j = 1, 2, ..., 2n-1$, whereif$X \subseteq B_j$,

$$\rho(X) = \begin{cases} \Delta_j \cdot \sum_{i=1}^{|X|} S_i & \text{if } |X| < m, \\ \Delta_j \cdot \sum_{i=1}^{m} S_i & \text{if } |X| \geq m \end{cases}$$

- $f(s,j) = f(A_j), j = 1, 2, ..., n$,
- $f(j,t) = f(B_j), j = 1, 2, 3, ..., 2n-1$

그림 2.2 : 스케쥴 가능 조건

J-node와 I-node사이의 아크(arc)는 J-node에 해당되는 태스크가 I-node의 시구간 사이에서 수행될 수 있음을 나타낸다. 주어진 태스크에 의한 network flow 그래프가 그림 2.2의 조건을 만족하는 경우에 스케쥴링이 가능하다.

그림 2.2에서(s, j)는 S노드에서 j번째 J-node로의 아크이고,(j,t)는 j번째 I-node에서 T노드로의 아크를 나타낸다. (j,i)는 j번째 I-node로의 아크를 나타낸다. A_j는 j번째 J-node에서 나간 아크의 집합이며, B_j는 j번째 I-node로 들어오는 아크의 집합이다. flow 함수 f는 각 아크의 가중치를 나타내는 함수이다. (1)과 (2)의 조건은 flow network사에서 나타난 계산요구시간과 처리기의 처리용량에 대한 제약조건이며, (3)과 (4)는 일관성을 위한 제약조건이다.

단일 처리기에서 주기적인 태스크(periodic task)의 스케쥴링은 EDF알고리즘을 쓸 수 있다. 그러나 EDF 알고리즘은 동적 우선순위 할당을 하고 있어 하드웨어 인터럽트를 효과적으로 처리할 수 없다. 시스템 정의 시간에 결정되는 하드웨어 인터럽트를 원활히 처리하기 위해서는 고정적인 우선순위가 주어지는 Rate-Monotonic 알고리즘이 적합하다[DL78].

1. Set $i = j = 1$
2. Assign task τ_i to processor P_j if this task together with the tasks that have been assigned to P_j can be feasibly scheduled on P_j according to the rate-monotonic scheduling algorithm for a single processor. If not, assign τ_i to P_{j+1} and set $j = j + 1$.
3. If $i < m$, set $i = i + 1$ and go to step 2; else stop.

그림 2.3 : next-fit 알고리즘

Rate-Monotonic 스케쥴링 알고리즘은 상대적으로 작은 주기를 가진 태스크에 높은 우선순위가 주어진다. 그러나 Rate-Monotonic 알고리즘은 EDF 알고리즘에 비하여 처리기의 사용효율(utilization)이 낮다. Rate-Monotonic 알고리즘으로 스케쥴링 하는 경우에는 처리기의 사용비율을 나타내는 $U = \sum \frac{Ci}{Pi}$가 0.7미만인 경우에만 스케쥴이 구성됨이 보장되나 EDF로 스케쥴링 하는 경우에는 1이하이면 스케쥴을 구성할 수 있다.

Rate-Monotonic 스케쥴링 알고리즘을 이용한 다중처리기 스케쥴링 알고리즘으로는 next-fit 알고리즘과 first-fit 알고리즘이 있다 [DL78]. next-fit 알고리즘은 그림 2.3과 같다. next-fit 알고리즘은 처리기를 하나씩 채워나가는 방식의 할당방법이다. 그림 2.4의 first-fit알고리즘은 새로운 태스크의 할당을 현재 사용중인 모든 처리기에 대하여 조사한다는 점이 next-fit과는 다르다. first-fit과 next-fit알고리즘 모두 최적 처리기 개수의 2배 이상을 사용하게 되는데, 이는 Rate-Monotonic알고리즘 처리기 사용 효율이 원인이다. 고정 우선순위 할당에서 태스크의 수행주기와 계산시간을 분할하여 처리기의 사용효율을 높일 수 있는 방법이 Sha에 의해 연구되었다.

이제까지 알아본 스케쥴링 알고리즘들은 모두 태스크의 수행도중 다른 태스크에 의한 선점(preemption)이 허용되는 경우의 스케쥴링 알고리즘이었다. 이제부터 태스크간의 선점이 허용되지 않는 경우의 알고리즘을 살펴본다.

Step 1. *Set* $i = N = 1$
Step 2. *(a) Set* $j = 1$.
(b) If Task τ_i *together with the tasks that have been assigned to processor* P_j *can be feasibly scheduled on processor* P_j *according to the rate-monotonic scheduling algorithm, assign* τ_i *to* P_j *and go to step 3.*
(c) Set $j = j + 1$ *and go to step 2(b).*
Step 3. *If* $j > N$, *set* $N = j$.
Step 4. *If all tasks have been assigned, then stop; else* $i = i + 1$ *and go to step 2.*

그림 2.4 : frst-fit 알고리즘

비선점 스케쥴링 문제(nonpreemptive scheduling problem)는 제한적인 경우를 제외하고는 거의 NP-hard임이 증명되었다. 단일 처리기에서 임의의 준비시간과 제한시간을 가지는 태스크의 스케쥴링은 NP-hard임이 증명되었으며, 다중 처리기 상에서는 제한시간과 준비시간이 동일한 제한적인 경우에서도 NP-hard임이 알려졌다 [Ull76]. 그러나 모든 태스크들이 동일한 준비시간을 가지는 경우에는 단일 처리기상에서 EDF알고리즘으로 스케쥴링 될 수 있음이 Moore에 의하여 증명되었다. 또한 태스크들의 제한시간의 순서와 준비시간의 순서가 동일한 경우에 단일 처리기에서 스케쥴링할 수 있는 최적 스케쥴링 알고리즘이 Kise에 의하여 개발되었다 [Kis78]. 다중처리기에서는 태스크의 수행시간이 단위시간인 제한적인 경우에만 스케쥴링을 할 수 있는 알고리즘이 개발되어 있다.

수행순서 제한(precedence constraints)을 가지는 태스크의 스케쥴링 단일 처리기에서 태스크간에 수행순서가 정의될 때에는 LDF(Latest Deadline First) 알고리즘이 최적 알고리즘이다. [Law73]. LDF알고리즘은 태스크들 간에 정의된 수행순서 그래프에서 이미 스케쥴된 태스크는 그래프 상에서 지워나가며 후계 태스크가 없는 태스크들을 스케쥴 한다.

LDF알고리즘이 개발된 이후 Blazewicz에 의해 수행순서 제한이 주어질 때 선점 스케쥴링과 비선점 스케쥴링이 동시에 존재하는 것이 증명되었다[Bla74]. 선점 스케쥴링을 하는 경우에는 태스크들 간의 순서제한 위배되지 않도록 준비시간과 제한시간을 조정하여 EDF알고리즘으로 스케쥴링 할 수 있다. 다중 처리기 스케쥴링에서 태스크 들의 수행순서 제한이 임의의 형태로 주어지는 경우에는 선점과 비선점 모두 NP-hard이다. 수행순서 제한을 정의하는 그래프가 트리 형태로 주어지는 경우에 사용될 수 있는 알고리즘으로 HLF(Higrest Level First)알고리즘이 있다. HLF알고리즘은 2개의 처리기상에서 비선점 태스크를 스케쥴링 하는 최적 스케쥴링 알고리즘이다. 여기서 레벨(level)은 트리상의 태스크가 리프(leaf)에 이르는 패스(path)의 길이 이다. 스케쥴은 수행준비가 된 태스크 중에서 가장 큰 레벨을 가지는 태스크를 먼저 수행시키는 방식으로 구성된다. 그러나 HLF알고리즘은 3개 이상의 처리기에서는 최적 알고리즘이 아니다.

선점 스케쥴링 알고리즘으로는 reverse precedence tree를 이용한 알고리즘과 regular tree를 이용한 알고리즘이 있다i [MJ70]. Muntz와 Coffman에 의해 개발된 선점 스케쥴링 알고리즘은 reverse precedence tree를 이용하여 리프에 가까운 노드에 해당되는 태스크에 높은 우선순위를 주어 스케쥴을 구성한다. Gonzalez와 Johnson의 알고리즘은 regular tree를 이용하여 태스크에 해당되는 노드의 subtree의 잔여 계산시간에 따라 우선순위를 부여 한다 [GJ77].

2.2.2 분산시스템에서의 정적 스케쥴링 알고리즘

분산시스템에서의 정적 스케쥴링 알고리즘은 기존의 태스크 할당문제와 깊은 관련이 있다. 기존의 분산 시스템에서 태스크 할당은 처리기 사이에 균등한 부하(load)와, 태스크간의 교신 비용(communication cost)의 최소화를 목적으로 한다. 이러한 문제는NP-hard로 알려져 있으며, 개발된 할당방법들의 휴리스틱(heuristic)알고리즘의 형태이다.

일반적인 시스템에서의 할당 알고리즘으로 Stone에 의해 연구된 network flow를 이용한 할당 알고리즘과 Bokari에 의해 개발되어진 할당 알고리즘들은 휴리스틱 함수에 태스크가 제한시간을 만족할 수 있도록 태스크의 수행 순서를 제한하는 휴리스틱 요소를 첨가하여 실시간 시스템에 사용할 수 있다. 그러나 상기의 알고리즘들은 통신비용의 최소화만을 고려하고 있으며, 부하 균등화(load balancing)는 고려하고 있지 않다. Efe 알고리즘의 휴리스틱 함수는 처리기의 부하를 균등하게 조절하면서 교신 비용을 최소화하는 휴리스틱 요소로 구성되어 있다[Efe82]. 알고리즘의 구성은 2단계로 이루어지는데, 1단계에서는 태스크간의 통신비용이 최소화될 수 있도록 태스크들을 클러스터(cluster)로 구성하고, 2단계에서는 각 클러스터에 속한 태스크들의 부하가 균등하게 되도록 재구성하여 처리기에 할당한다. Efe의 알고리즘 역시 실시간 시스템에 적용이 가능하다. 기존의 할당 알고리즘을 이용하는 방법이외에 스케쥴링 알고리즘의 구성방법으로는 정수 프로그래밍(integer programming)모델을 이용한 것이 있다.

2.3 동적 실시간 스케쥴링 알고리즘

본 절에서는 동적 실시간 스케쥴링 알고리즘에 관하여 알아본다. 동적 실시간 스케쥴링 알고리즘은 태스크의 준비시간, 계산시간, 제한시간과 같은 정보들이 태스크의 발생이후에 시스템에 알려질 때 사용하는 알고리즘이다. 동적 실시간 시스템은 정적 실시간 시스템에 비하여 보다 높은 적응성(adaptability)과 유연성(flexibility)이 보장되므로 시스템의 모든 동작을 미리 예측할 수 없는 복잡한 시스템의 스케쥴러의 구현에 적합한 알고리즘이다.

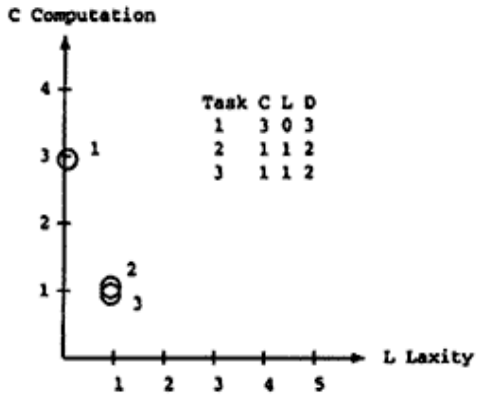


그림 2.5 : L-C 평면

2.3.1 중앙 집중 시스템에서의 동적 실시간 스케쥴링 알고리즘

태스크의 수행도중 다른 태스크에 의한 선점이 허용되는 경우, 단일 처리기상에서 EDF알고리즘과 LLF알고리즘이 최적 스케쥴링 알고리즘이다. 다중 처리기에서는 최적 스케쥴링 알고리즘이 존재할 수 없음이 증명되었다. 증명은 그림 2.5의 L-C 평면을 이용하여 현재의 태스크에 대하여 어떤 스케쥴링 결정이 내려지더라도 그것이 최적 스케쥴링일 수 없도록 하는 태스크가 항상 동적으로 발생할 수 있음을 보였다.

또한 Mok은 L-C 평면상의 태스크들을 다음과 같이 3종류로 구분하고

$R_1(k) = \{J_i : D_j \leq k\}$

$R_2(k) = \{J_i : L_i \leq k \text{ and } D_j > k\}$

$R_3(k) = \{J_i : L_i > k\}$

- Minimum deadline first (Min_D): $H(T) = T_D$;
- Minimum processing time first (Min_P): $H(T) = T_P$;
- Minimum earliest start time first (Min_S): $H(T) = T_{est}$;
- Minimum laxity first (Min_L): $H(T) = T_D - (T_{est} + T_P)$;
- Min_D + Min_P : $H(T) = T_D + W * T_P$;
- Min_D + Min_S : $H(T) = T_D + W * T_{est}$;

그림 2.6 : Myopic 알고리즘의 휴리스틱 함수

잉여 계산시간 함수

$$F(k) = k \ \text{X} \ n - \sum_{R_1} Cj - \sum_{R_2} (k - Lj), k > 0$$

가 0보다 크거나 같은 경우에는 LLF알고리즘으로 스케쥴링 가능함을 증명하였다.

비선점 다중 스케쥴링 알고리즘으로 휴리스틱 알고리즘인 Myopic 알고리즘이 있다. Myopic 스케쥴링 알고리즘의 휴리스틱 함수는 그림 2.6에 나타난 바와 같이 태스크의 준비시간, 계산시간, 제한시간 등이 고려되고 있다. Myopic 알고리즘의 주요 자료구조로는 태스크나 자료의 가용시간을 나타내는 EAT 리스트가 있다. Myopic알고리즘은 EAT리스트 상에서 현재 수행 가능한 태스크를 결정하고 이중 최적의 휴리스틱 함수 값을 지닌 태스크를 수행시킨다.

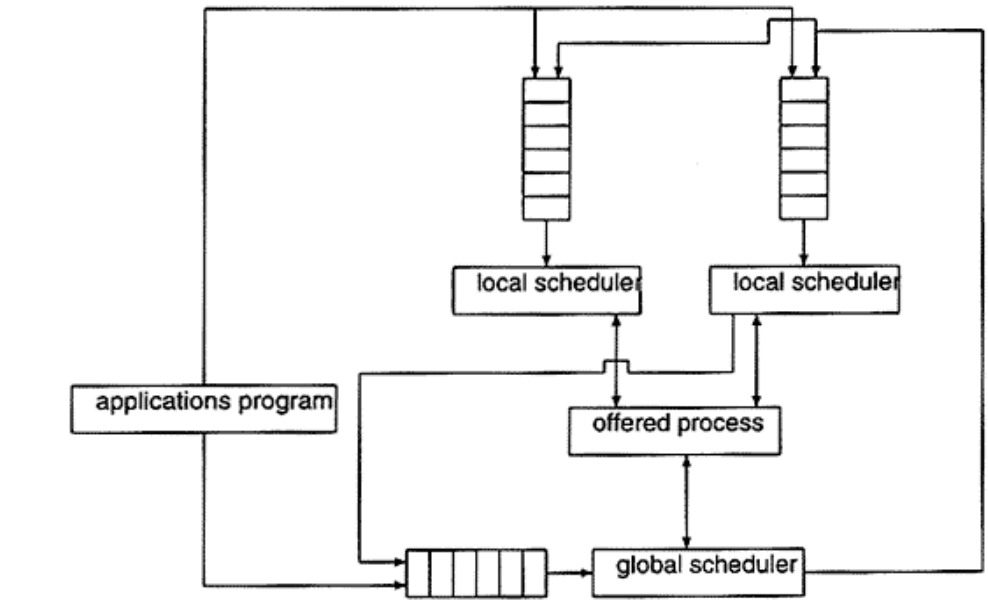


그림 2.7 : 분산 시스템에서의 실시간 스케줄러의 구성

2.3.2 분산 시스템에서의 동적 실시간 스케쥴링 알고리즘

분산 실시간 시스템에서의 동적 스케쥴링 알고리즘은 그림 2.7과 같이 지역 스케쥴링 알고리즘(local scheduling algorithm)과 전역 스케쥴링 알고리즘(global scheduling algorithm)으로 구성된다. 지역 스케쥴링 알고리즘의 역할은 지역 노드(local node)에서 발생한 태스크를 스케쥴링 하고, 잉여 계산시간이 허용되면 전역 스케쥴러에서 전달된 태스크를 처리하는 것이다. 전역 스케쥴링 알고리즘은 지역 노드에서 처리될 수 없는 태스크를 다른 노드로 보내거나. 다른 노드에서 보내온 태스크를 스케쥴링 할 수 있는지를 점검하는 역할을 한다. 지역 스케쥴링 알고리즘은 중앙 집중 시스템의 스케쥴링 알고리즘을 사용할 수 있다.

전역 스케쥴링 알고리즘은 항상 네트워크상의 노드들에 관한 정보를 유지하여야 한다.

그러나 시스템의 상태는 시간에 따라 동적으로 변화하므로 현재의 정보가 정확한 것일 확률은 낮다. 이러한 부정확한 전역 정보를 바탕으로 보다 높은 스케쥴링 가능성을 얻기 위한 전역 스케쥴링 알고리즘으로는 focused addressing bidding의 혼용 알고리즘이 개발되어 있다. 알고리즘의 구조는 그림 2.8과 같다.

2.4 결함 허용성 실시간 스케쥴링 알고리즘(Fault Tolerant Real Time Scheduling A1-gorithm)

결함 허용성 실시간 시스템은 시스템에 결함(fault)이 발생하더라도 일반적인 시스템과 마찬가지로 지속적인 동작을 하면서 정의된 시간제한을 준수하여야 한다. 그러므로 결함 허용성 실시간 스케쥴링 알고리즘은 결함에 의해 발생한 오류(error)의 복구(recovery)처리를 주어진 시간제한에 맞춰서 실행할 수 있도록 스케쥴링 하는 기능을 지원해야 한다. 본 장에서는 결함 허용기법이 적용된 실시간 스케쥴링 알고리즘에 관하여 살펴본다.

2.4.1 단일 처리기 결함 허용성 스케쥴링 알고리즘

Liestman과 Campbell은 단일 처리기에서 복구블럭(recovery block)을 가지는 주기적인 태스크를 스케쥴링 하는 알고리즘을 개발하였다[LC86]. 복구 블록이란 주 태스크(primary task)가 실패한 경우, 이를 대체하는 태스크이다. Liestman과 Campbell의 알고리즘은 태스크들의 주기가 그림 2.9과 같은 단순 주기인 경우에 사용될 수 있는 알고리즘으로 스케쥴링 되는 주태스크의 개수를 최대화하는 결함 허용 최적(fault tolerant optimal)알고리즘이다.

The Focussed Addressing Scheme:

```
/* The following code is executed by the local scheduler task
   on a node whenever a local task cannot be guaranteed. */

   Let a task to be sent out be a 3-tuple,(C,D,Size), where
C = computation time, D = deadline, Size = seze of task in packets;
   Let node_surplus[1..n] be the information available on this node
about the percentage of free time on nodes 1 througе n;
   Let FP be the adaptive parameter used in focussed addressing;

Begin
For each remote site i
  Do Estimate ART, the arrival time if the task at node i,
  should be task be sent to node i;
If node_surplus[i]*(D-ART) > FP*C
then mark node i as a candidate for a focussed node;
  End Do;

If there is any candidate for the focussed node
then begin
Randomly choose one of the candidate to be the focussed node;
Let the chosen focussed node be node j;
Send the task to node j;
Store task in bidder's queue of tasks for which Request
  For Bids have to be sent out (Indicate that bids should be
  returned ti the focussed node j);
     End
else
     Store task in bidder's queue of tasks for which Request
  For Bids have to be sent out (Indicate that bids should be
  returned ti the focussed node j);
End If;
End;
```

그림 2.8 : Focused Addressing 17

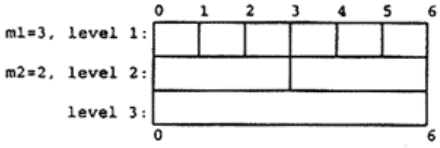


그림 2.9: 단순 주기

1. Construct a provisional schedule S_{i+1} by concatenating m_i copies of schedule S_i
2. Modify S_{i+1}. Remove the minimum number (optimality requirement) of largest P_k's for $1 \leq k \leq i$, increasingly idle time until A_{i+1} can be scheduled. Schedule A_{i+1}
3. If the resulting idle time is large enough, then schedule P_{i+1}.
4. Otherwise, if one exists, find the largest scheduled primary P_k, $i \leq k \leq i$, If $P_k > P_{i+1}$, then remove P_k and schedule P_{i+1}

그림 2.10: 알고리즘 1

Liestman과 Campbell이 개발한 스케쥴러는 3개의 알고리즘으로 구성된다. 알고리즘 1은 그림 2.10에 나타난 바와 같이 작은 주기를 가지는 태스크로부터 큰 주기를 가지는 태스크의 순서로 스케쥴을 구성하면서 스케쥴링되는 주 태스크의 개수가 최대한 많아지도록 한다.

그림 2.10에서 S_i는 태스크 J_1에서 태스크 J_i까지의 스케쥴이다. P_i는 태스크 J_i의 주 태스크이고 A_i는 태스크 J_i의 복구 블록이다.

$$A_1 = 3, P_1 = 6, T_1 = 10.$$

$$A_2 = 3, P_2 = 3, T_2 = 20.$$

$$A_3 = 2, P_3 = 5, T_3 = 40.$$

그림 2.11 : 단순 주기 태스크의 예

알고리즘 1은 작은 주기의 태스크부터 차례로 스케쥴링 하면서 전체 태스크를 전부 스케쥴링 할 때까지 각 태스크 마다 한번씩 수행된다. 알고리즘은 매번 수행될 때마다 이전의 스케쥴을 현재 스케쥴링 하는 태스크의 주기에 맞춰서 나열한 후에 현재 태스크의 복구블럭과 주태스크를 스케쥴링 한다. 첫 번째로 구성되는 스케쥴 S_1은 가장 작은 주기의 태스크를 자신의 주기 내에 스케쥴 한 것이다.

그림 2.11과 같이 단순주기의 태스크가 주어질 때의 스케쥴링 과정을 살펴본다. 그림 2.11에서 T_i는 J_i의 주기이다. 그림 2.12의 (a)는 P_1으로 S_1을 구성한 이후, P_2의 주기에 맞게 S_1을 2개로 복사하고, 알고리즘1의 단계2에서 $A2$의 스케쥴을 위해 두 번째 P_i를 삭제한 상태이다. 그림 2.12(b)는 두 번째 태스크를 스케쥴링한 S_2이다. 이러한 과정은 세 번째 태스크에도 마찬가지로 적용되어 최종적으로 그림 2.12의(d)의 S_3가 구성된다.

알고리즘 1에 의해 구성된 스케쥴에서 주태스크가 종료된 경우 해당 주기에 스케쥴된 복구블럭은 수행될 필요가 없다. 그러므로 복구블럭이 수행되지 않도록 스케쥴을 재구성할 필요가 있다. 그림 2.13과 같이 구성된 스케쥴에 의해 수행되는 도중 두 번째 P_1이 수행을 성공적으로 종료 하였을 때, 같은 주기에 있는 A_1은 수행될 필요가 없다. 그러므로 14에서 60까지의 시구간사이의 스케쥴을 재구성되어야 한다. 알고리즘 2는 이러한 경우 나머지 구간을 재구성하는 알고리즘이다.

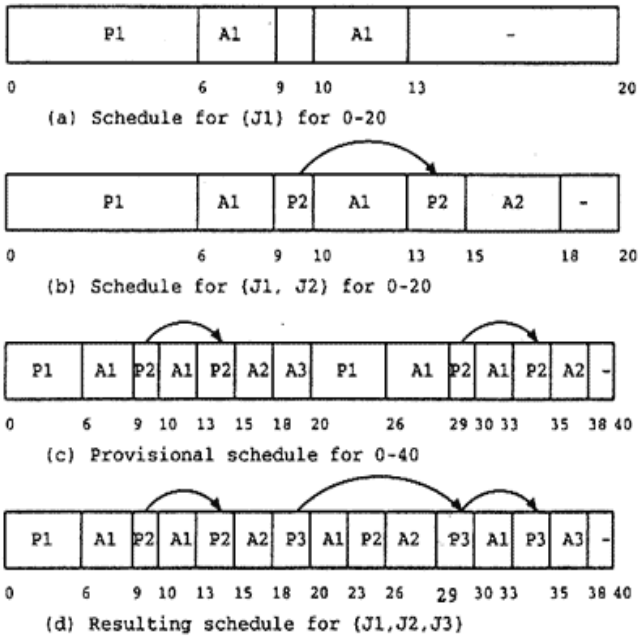


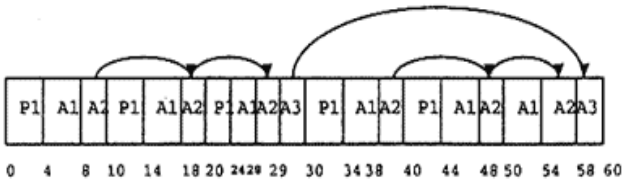
그림 2.12: 알고리즘 1의 스케쥴링 과정

A1 = 4, P1 = 4, T1 = 10,

A2 = 5, P2 = 7, T2 = 30,

A3 = 6, P3 = 8, T3 = 60.

(a) three jobs to be scheduled dynamically



(b) An FT Schedule for period 0 - 60

그림 2.13: 스케쥴링의 사례

Algorithm 2 : As before, the schedule is created iteratively, beginning at the lowest level.
1) For level i, $i \neq r$ create two schedules, SHORT_i and FULL_i. SHORT_i is the schedule for the remainder of the current period T_i, and FULL_i is the schedule for the future T_i periods in the period T_r.
2) For level r, create only the SHORT_r schedule.
3) SHORT_i is constructed for the interval g_s to D_i by concatenating SHORT_{i-1} for the interval g_s to D_{i-1} with $(D_i - D_{i-1})/T_{i-1}$ copies of FULL_{i-1} for $i > 1$.
4) FULL_i is constructed by modifying m_{i-1} copies of FULL_{i-1}, as in Algorithm 1.

그림 2.14 : 알고리즘 2

알고리즘 2의 구조는 그림 2.14와 같다. 알고리즘 2에서 $EX\ A_i$는 임의의 태스크P_s가 종료하였을 때 A_i가 이미 수행된 시간이다. EXP_i는 P_i가 이미 수행한 시간이다. R_i는 P_s가 종료한 시점에서 J_i의 제한시간 D_i까지의 잔여시간이다.

알고리즘 3은 알고리즘 1과 알고리즘 2를 이용하여 그림 2.15와 같이 결함이 발생한 경우, 이를 처리할 수 있는 스케쥴을 트리구조의 스케쥴 테이블을 구성한다. 트리의 루트는 알고리즘 1과 알고리즘 2에 의하여 구성된 모든 주태스크가 성공한 경우의 스케쥴이다. 트리의 리프는 주 태스크가 실패한 경우를 위한 스케쥴로 구성된다.

2.4.2 다중 처리기 결함 허용성 스케쥴링

다중 처리기에서 복구 블록 형태의 태스크를 스케쥴링하는 알고리즘으로는 알고리즘 Q가 있다. [KS86]. 알고리즘 Q는 기존의 할당 알고리즘과 스케쥴링 알고리즘을 이용하여 구성된 알고리즘으로, 다음의 조건이 만족되도록 주태스크와 복구 블록을 처리기에 스케쥴링 한다.

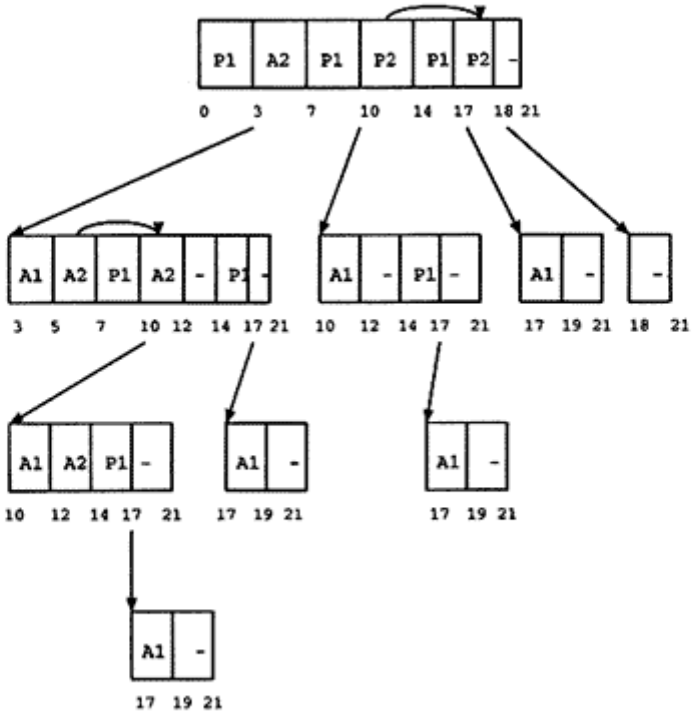


그림 2.15 : 알고리즘 3에서 구성된 스케쥴 트리

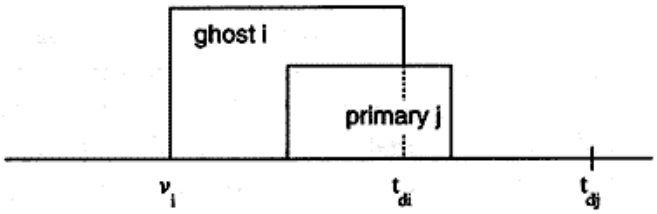
・조건 1 : 같은 태스크의 주 태스크와 복구블럭은 동일 처리기에 할당될 수 없다. ・조건 2 : 그림 2.16에서와 같이 주태스크와 복구블럭이 겹치게 스케쥴 될 때는 주태스크와 복구블럭이 모두 제한시간을 만족할 수 있어야 한다.

알고리즘 Q의 구조는 그림 2.17과 같다. 그림 2.17에서 P_1은 기존의 할당 알고리즘 이다 P_2는 기존의 단일 처리기 스케쥴링 알고리즘이며, P_2^*는 P_2를 변환하여 홀에 복구블럭을 채우도록 만든 알고리즘이다. 홀이란 주태스크가 스케쥴 되지 않은 구간이다. π_i와 Θ_i는 i처리기에 할당된 주태스크와 복구 블록의 집합이다.

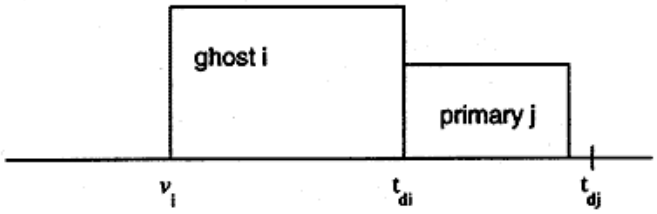
2.5 결 론

본 장에서는 기존의 실시간 스케쥴링 알고리즘에 대하여 조사하였다. 정적 스케쥴링 알고리즘에서는 주로 주기적인 태스크를 위한 스케쥴링 알고리즘이 많이 개발되어 있었다. 이러한 정적 알고리즘은 실시간 시스템에서 많이 발생하는 주기적인 태스크를 효율적으로 스케쥴링 할 수 있는 알고리즘으로 실시간 시스템의 설계 및 실시간 태스크의 우선순위 결정에 이용된다. 정적 분산 알고리즘은 분산망 상의 노드에 실시간 태스크를 할당하는 알고리즘으로 휴리스틱 알고리즘이 개발되고 있다. 동적 스케쥴링 알고리즘은 비주기적으로 발생하는 실시간 태스크를 스케쥴링하는 알고리즘으로 정적 알고리즘에 비하여 실행시간 부하가 상대적으로 큰 단점이 있으나 flexiblity가 좋다.

고장 허용성 실시간 스케쥴링 에서는 주로 timing redundancy를 이용한 방법이 사용 되고 있다.

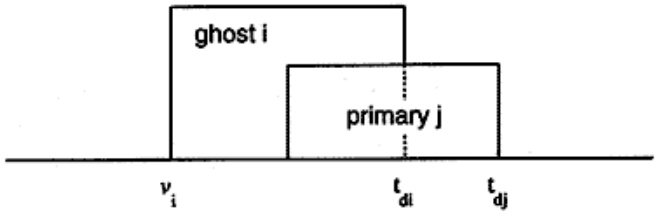


Overlapped schedule



Ghost i Activated

(a)



(b)

그림 2.16 : 조건 2의 예

Algorithm Q is as follows. Let the set of primaries and ghosts allocated to i be π_i and θ_i respectively. For every allocation of primaries and ghosts to processors by P_1 do Step 1–6 for every processor i:
Step 1) Record the holes for processor i, defined by the allocation.
Step 2) Run $P_2^*(\pi_i \cup \theta_i)$. Return control to P_1 if the schedule is found to be infeasible. Otherwise, record the positions of the ghosts in G.
Step 3) Set $j := 0$, and, while θ_i is nonempty, do a)–f).
a) Initialize P_2 at $\nu_{\text{ghost}_i(j)}$. (* See definition of initialization above. *)
b) Set $\theta_i := \theta_i - \{\text{ghost}_i(j)\}$.
c) Transform the hard deadlines of the primary clones $k \in \pi_i$ to be $t_{dk}^{(0)} - \text{util}_{\theta_i}^G(0, t_{dk}^{(0)})$, where $t_{dk}^{(0)}$ is the original hard deadline of clone k and $\text{util}_{\theta_i}^G(a, b)$ is the nonprimary-hole space occupied by the ghosts in θ_i, according to the ghost schedule G. (* Guarantees there is enough time redundancy for all deadlines to be met if one or more ghosts in G are activated. *)
d) Run $P_2(\pi_i)$ with these new deadlines, and let S_{ij} be the schedule obtained. (* S_{ij} is the contingency schedule, a portion of which will be used to obtain the optimal schedule. *)
e) Calculate the remaining run time of the clones scheduled in d) by subtracting from the current remaining run time the duration for which they were scheduled in the interval $[\nu_{\text{ghost}_i(j)}, \nu_{\text{ghost}_i(j+1)}]$. f) Set $j := j + 1$.
Step 4) If all primary clones $k \in \pi_i$ have not been completed by $\nu_{\text{ghost}_i(j)}$, set $t_{dk} := t_{dk}^{(0)}$, initialize P_2 at $\nu_{\text{ghost}_i(j)}$, and run $P_2(\pi_i)$, recording the output in S_{ij}.
Step 5) Form schedule S_i by piecing together the schedules represented by S_{ij} in the interval $[\nu_{\text{ghost}_i(j)}, \nu_{\text{ghost}_i(j+1)}]$. Letting $\{C_k\}$, $k \in \pi_i$ be the completion times of the primary clones k according to schedule S_i, compute $\sum_{k \in \pi_i} f_k(C_k)$, and return this value as the cost associated with S_i. (* See below for explanation. *)
Step 6) Restore all hard deadlines of the primary clones $k \in \pi_i$ to their original values $t_{dk}^{(0)}$.

그림 2.17 : 알고리즘 Q

태스크가 수행도중 소프트웨어 오류나 일시적인 하드웨어 결함에 의하여 실패하게 되는 경우 복구 블록이 수행된다. 이러한 복구 블록은 시간제한이 엄격하게 주어질 수도 있고, 연성으로 주어질 수 도 있다. 이는 발생된 고장의 종류에 따라 달라지는 것이다. 또한 복구블럭의 수행은 고장의 발생에 따라 결정되는 것으로 동적으로 발생하게 된다.

실시간 스케쥴링 알고리즘은 이상에서 알아본 바와 같이 응용 의존적으로 개발되고 있으며, 동적으로 발생하는 고장을 처리하기 위하여 복구블럭을 정의된 시간제한에 따라 처리하는 기법들이 연구되고 있다. 그러므로 차세대 전 전자 교환기를 위한실시간 운영체제의 스케쥴링 알고리즘의 개발은 전 전자 교환기의 시스템 특성에 따라 설계, 개발되어야 한다.

제 3 장 교환기의 운영체제에서의 실시간 처리 요구분석

본 장에서는 차세대 전 전자 교환기의 스케쥴러 구현을 위하여 기존의 전 전자 교환기 시스템 운영체제의 특성과 차세대 전 전자교환기 운영체제의 요구사항을 알아보고, 이를 바탕으로 하여 차세대 전 전자교환기의 스케쥴러의 구현을 위한 알고리즘의 특성을 정리한다.

3.1 교환기 운영체제의 구조 분석

전 전자 교환기 시스템의 기본적인 구조는 그림과 같이 계층 구조를 가진 분산 시스템으로 각 노드는 전용된(dedicated)기능을 수행하고 있다.

연결망에 접속된 지역 노드내의 시스템 구조는 그림 3.2와 같이 상주 프로세스(resident process)가 외부의 요구(request)에 의해 이를 처리하는 자 프로세스(child process)를 발생시키는 사건 주도형(event driven)이며, 태스크들의 우선순위는 고정 우선순위(fixed priority)이다. 태스크에 주어지는 시간제한은 연성 제한시간(soft dead-line)이며, 태스크를 수행하기 위한 자원에 대한 요청은 대기 한계시간(timing bound)을 가진다. 만약 이 한계시간이 초과 된 경우에는 현재의 요청 태스크는 실패한 것으로 간주되며, 일정 시간 이후 동일 태스크가 재시도된다.

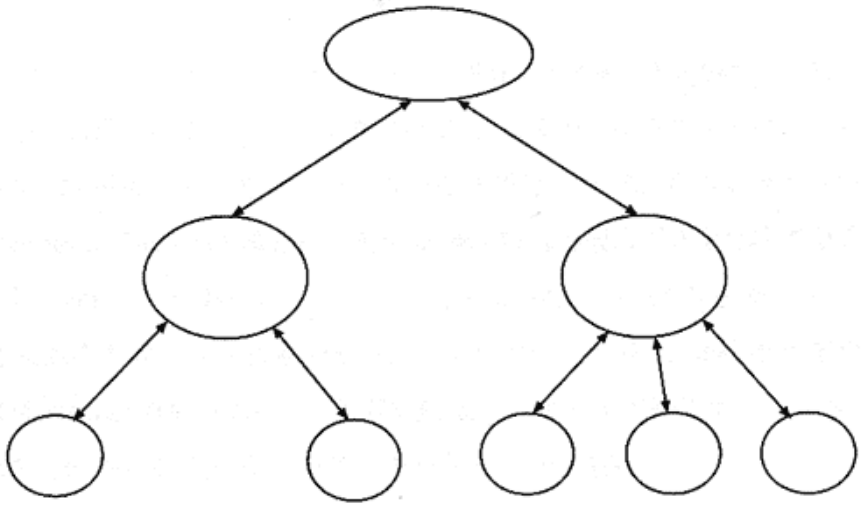


그림 3.1: 교환기 시스템의 기본적인 구조

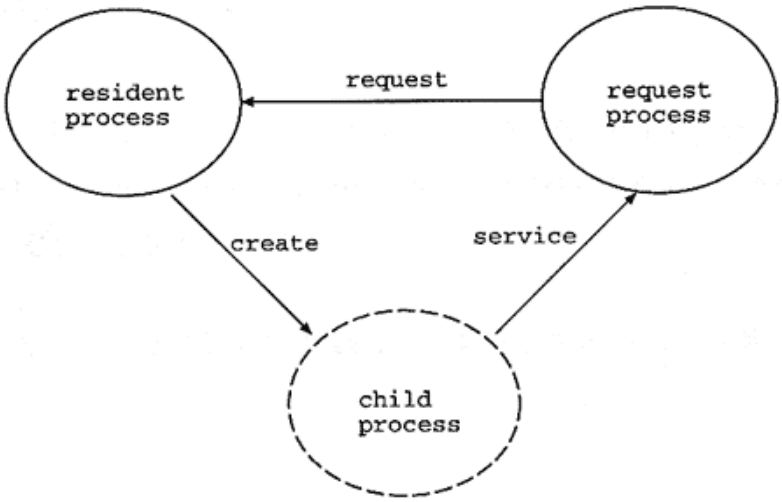


그림 3.2: 시스템의 구조

3.2 차세대 전 전자교환기 운영체제의 요구사항

차세대 전 전자 교환기는 기존의 공중 전화망에서의 서비스에 국한하지 않고 BISDN(Broadband ISDN)상에서의 서비스도 고려하여 설계되어야 한다. BISDN에서는 정확하고 신뢰성 높으며, 시간적 투명성을 가지는 전달 모드를 통하여 POTS(Plain Old Telephone Service)지원뿐만 아니라, 데이터 통신 및 TV전송 등의 복합된 기능을 지원한다. 그러므로 차세대 전 전자 교환기의 운영체제는 B-IDSN을 통하여 이루어 질 수 있는 새로운 형태의 서비스를 지원하는 기능을 갖추어야 한다. 특히 정보 전달의 시간적 투명성(timing transparency)은 다양한 실시간 응용을 가능하게 하므로 차세대 전 전자 교화기는 실시간 처리를 지원하는 기능을 갖추어야 한다.

3.3 차세대 전 전자 교환기를 위한 스케쥴링 알고리즘의 요구사항

본 절에서는 기존의 전 전자 교환기와 차세대 전 전자 교환기에서 나타난 운영체제상의 특성을 통해 스케쥴링 알고리즘에 요구되는 기능을 정리하고 기존의 스케쥴링 알고리즘에 적용가능성을 점검한다.

차세대 전 전자 교환기를 위한 스케쥴링 알고리즘의 요구사항을 정리하면 다음과 같다.

・다양한 시간제한(timing constraints)을 가지는 태스크의 처리: 차세대 전 전자 교환기에서는 BISDN을 통하여 TV전송과 멀티미디어 서비스의 지원 등이 가능해야 하므로 비 주기적인 연성 실시간 태스크 이외에도 경성 실시간 태스크도 지원해야 한다.

・고정우선순위를 가지는 태스크 : 교환기 운영체제에서의 태스크들은 앞서 살펴본 바와 같이 모프로세스의 우선순위를 그대로 이어 받아 동작하게 된다. 그러므로 태스크에 대한 우선순위 할당은 고정적이어야 한다.

・결함 허용성(fault tolerance)지원 : 전 전자 교환기의 태스크들은 대부분이 연성 실시간 제한을 가진다. 그러므로 복구블럭의 수행은 응답시간이 최소화 되어야 한다.

본 연구에서는 위의 요구사항을 고려한 실시간 스케쥴링 알고리즘을 개발한다.

제 4 장 제안된 실시간 스케쥴링 알고리즘

4.1 서 론

실시간 시스템(real-time system)은 시스템에 정의된 시간제한(time constraints)에 따라 논리적으로 정확한 동작을 하도록 설계된 시스템으로, 항공기 제어 시스템이나 공정 제어 시스템과 같은 예가 있다[SR88]. 이러한 시스템을 구성하는 태스크는 대부분이 제어 대상 시스템을 감시(monitoring)하는 주기 태스크(periodic task)이다 [Koo92]. 주기 태스크의 스케쥴링(scheduling)은 실시간 태스크 스케쥴링의 가장 고전적인 분야로서, 현재도 많은 연구가 진행되고 있다 [Koo91].

주기 태스크를 스케쥴링 하는 알고리즘은 우선순위(priority)의 할당 방법에 따라, 고정 우선순위(fixed priority)알고리즘과 동적 우선순위(dynamic priority)알고리즘으로 분류된다. [LL73]. 고정 우선순위 알고리즘은 한 태스크의 모든 인스탄스(instance)가 동일한 우선순위를 가지는 스케쥴링 방법이고, 동적 우선순위 알고리즘은 이와 달리 각 인스탄스가 다른 우선순위를 가지는 스케쥴링 방법이다 [LL73]. 동적 우선순위 알고리즘으로는 EDF(Earliest Deadline First)알고리즘 [LL73]과 LSF(Least Slack First)알고리즘[MD78]이 있으며, 고정 우선순위 알고리즘으로는 RM(Rate Monotonic)알고리즘[LL73, DL78]과 DM(Deadline Monotonic)알고리즘[Leh90]이 있다.

동적 우선순위 알고리즘은 정적 우선순위 알고리즘에 비하여 이론적으로는 스케쥴링 가능성(schedulability)이 우수한 반면에, 정적 우선순위 알고리즘에 비하여 스케쥴링의 오버헤드(overhead)가 크다.

이와 달리 정적 우선순위 스케쥴링은 이론적으로 스케쥴링 가능성은 낮지만, 실질적으로는 평균 처리기 사용도(usability)가 88%정도로 실용성이 높아서 실시간 시스템의 설계에는 구현이 복잡한 동적 우선순위 알고리즘보다는 상대적으로 간단한 정적 우선순위 알고리즘이 많이 이용되고 있다[Koo91]. 그러나 보다 신뢰성 있는 제어 시스템의 설계를 위해서는 인터럽트(interrupt)나 복구 처리(recovery processing)등에 의하여 발생하는 비주기 태스크(aperiodic task)의 효율적인 처리가 요구되어, 고정 우선순위 알고리즘에 의하여 설계된 실시간 시스템에서 동적으로 발생하는 비주기 태스크를 효율적으로 처리하기 위한 스케쥴링 방법이 필요하게 되었다. 이러한 태스크 스케쥴링을 혼합 태스크 스케쥴링(mixed task scheduling)이라 한다 [LS92]. 혼합 태스크 스케쥴링의 목적은 고정 우선순위 시스템에서 동적으로 발생하는 비주기 태스크의 응답 시간(response time)을 최소화하는 것이다. 응답 시간의 최소화를 위해서는 주기 태스크의 슬랙 타임(slack time)을 최대한 이용하는 것이 요구되며, 이는 비주기 태스크의 시간제한 특성에 관계없이 기본적으로 충족되어야 하는 문제이다.

주기적인 태스크와 비주기적인 태스크를 동시에 스케쥴링 하는 혼합 태스크 스케쥴링 방법 중에서 가장 직관적이고 간단한 방법은 비주기적인 태스크를 후위 처리(background)하는 방법이다. 그러나 후위 처리를 하는 경우에는 주기 태스크가 항상 비주기 태스크를 우선하므로 응답 시간의 최소화를 기대할 수 없다. 보다 개선된 방법으로는 비주기적인 태스크를 처리하기 위한 주기 태스크를 별도로 설정하는 방법이다. 그러나 이 경우에는 비주기적인 태스크가 해당 주기에 발생하지 않는 경우에는 예약된 시간이 소모될 수 있다. 이를 개선하기 위한 연구의 결과로 PE(Priority Exchange)알고리즘이 개발되었다 [Leh87]. 그러나 PE알고리즘은 주기 태스크의 시작 부분에서 비주기 태스크가 감지되지 못하면 다음 주기까지 기다려야 하는 단점이 있다. 예약된 주기 동안 지속적으로 비주기 태스크를 서비스 할 수 있는 알고리즘으로는 스포라딕(sporadic) 서버 알고리즘과 디퍼러블(deferrable) 서버 알고리즘이 있다 [Spr89]. 이상과 같은 알고리즘들을 폴링 서버(polling server) 방식이라고 한다. 폴링 서버 방식은 비주기 태스크의 도착시기가 항상 주기와는 일치하지 않는 단점이 있어 동적으로 발생하는 비주기적인 태스크의 서비스에는 한계가 있다.

최근의 연구로는 슬랙 스틸링(slack stealing) 서버 알고리즘이 있다 [LS92]. 슬랙 스틸링 서버 알고리즘은 RM 알고리즘에 의하여 주기 태스크에 우선순위를 할당한 상태에서 비주기 태스크를 처리하는 알고리즘은 폴링 서버 방식과는 달리 주기적인 폴링 태스크를 사용하지 않고 비주기 태스크가 발생하는 경우 슬랙 스틸링 서버가 주기 태스크의 제한시간을 보장(guarantee)하면서, 이용 가능한 최대 슬랙 시간을 계산하여, 이 시간을 비주기 태스크의 수행에 이용하는 방식이다. 그러나 슬랙 스틸링 서버 알고리즘이 이용하고 있는 RM 스케쥴링은 비주기 태스크에 의한 주기 태스크의 긴급도 변화에 적절히 적응하지 못하는 단점이 있다.

본 연구에서는 RM 알고리즘과 동일한 스케쥴링 가능성을 가지는 새로운 고정 우선순위 스케쥴링 알고리즘을 제안하고, 이를 이용하여 기존의 슬랙 스틸링 알고리즘을 개선하고자 한다. 새로운 고정 우선순위 알고리즘의 가장 큰 특징은 태스크에 따라 2개의 우선순위가 할당된다는 것에 있다. 이러한 이중 우선순위는 슬랙 스틸링 알고리즘이 비주기 태스크를 위한 슬랙 시간을 보다 효율적으로 얻어낼 수 있도록 하여 비주기 태스크의 응답시간을 개선할 수 있도록 해준다.

본 연구에서는 이중 우선순위 알고리즘이 RM 알고리즘과 같은 스케쥴링 가능성을 가짐을 증명함으로서, 이중 우선순위 알고리즘이 RM 알고리즘을 대체할 수 있음을 입증한다. 또한 이중 우선순위 알고리즘에 의한 고정 우선순위에 따라, 슬랙 스틸링 서버 알고리즘을 적용할 경우, RM 알고리즘에 따른 슬랙 스틸링 보다 비주기 태스크의 응답시간이 개선됨을 보인다.

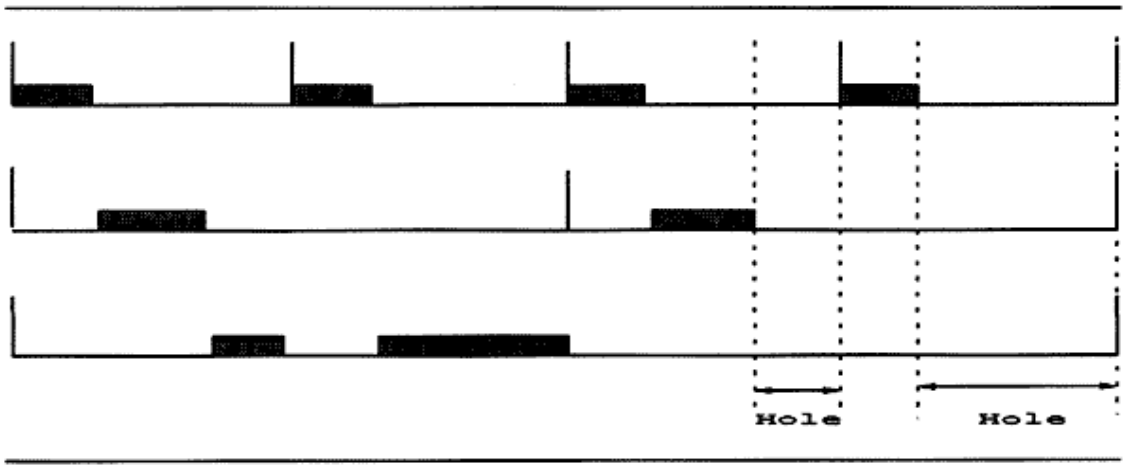


그림 4.1 : Hole

4.2 기존의 스케쥴링 알고리즘

본 절에서는 고정 우선순위 시스템에서 비주기 태스크를 스케쥴링 하는 기존의 스케쥴링 알고리즘에 대하여 알아보고, 이들의 문제점을 논한다.

4.2.1 폴링 서버(polling server)

고정 우선순위를 가지는 주기 태스크와 동적으로 발생하는 비주기 태스크를 스케쥴링 하는 방법으로 가장 간단한 것은 비주기 태스크를 그림 4.1에서와 같이 주기 태스크를 실행하고 남는 시간인 비 스케쥴링 구간인 hole을 이용하여 후위 처리하는 방법이 있다. 그러나 후위 처리 방식에서는 현재 준비 중인 주기 태스크가 모두 수행된 이후에 비주기 태스크가 처리되므로 비주기 태스크의 응답시간이 좋지 않다. 후위 처리보다 개선된 방식인 폴링 서버 방식은 비주기 태스크를 처리하기 위한 폴링 태스크를 주기적으로 실행시키면서 비주기 태스크를 위한 계산시간을 예약함으로써 비주기 태스크의 응답시간을 개선하는 방식이다. 이러한 계산시간의 예약은 비주기 태스크가 도착하지 않는 경우에는 소모

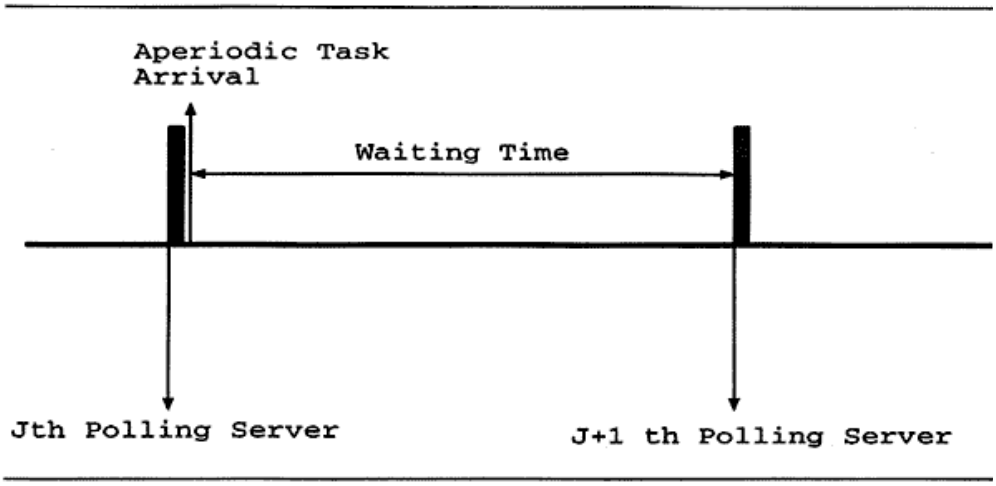


그림 4.2 : PE알고리즘에서 발생하는 최악의 대기시간

될 수 있으므로, 준비된 비주기 태스크가 없는 경우에 주기 태스크에 계산시간을 양도하는 PE알고리즘이 개발되었다 [Leh87].

비주기 태스크의 발생이 폴링 태스크의 주기와 일치하지 않으면서 발생될 때의 또 다른 문제점은 PE알고리즘에 의하여 폴링 태스크의 계산시간이 다른 주기 태스크에 양도된 이후에 비주기 태스크가 발생하였을 때 생기게 된다. 이 경우에는 다음번의 폴링 태스크까지 비주기 태스크의 대기시간이 늘어나게 되어, 그림 4.2와 같이 불필요한 대기 시간이 발생하게 된다. 이러한 원인은 폴링 서버의 동작이 자신의 주기 시작 시점이후에 발생할 수 있는 비주기 태스크에 대한 고려가 없다는 데에 있다. 이러한 문제점을 해결하기 위한 방법으로 스포라딕 서버(sproradic server)와 디퍼러블 서버(deferrable server)가 개발되었다 [Spr89]. 스포라딕 서버와 디퍼러블 서버는 폴링 태스크의 주기내에서는 언제든지 비주기 태스크의 도착에 따라 이를 처리할 수 있도록 개선된 알고리즘이다. 그러나 스포라딕 서버와 디퍼러블 서버 역시 폴링 태스크의 계산시간 한도 내에서만 비주기 태스크의 처리가 가능하며, 비주기 태스크의 계산 시간이 이를 초과하는 경우에는 다음 주기까지 대기하여야 한다.

4.2.2 슬랙 스틸링(slack stealing)

슬랙 스틸링 알고리즘 [LS92]은 엄격한 제한시간을 가지는 주기 태스크의 시간제한을 준수하면서 비주기 태스크의 응답시간을 최소화하기 위하여 폴링 서버 방식과는 달리 주기적으로 동작하는 서버를 이용하지 않고 비주기 태스크가 발생하였을 때, 이를 수행시키기 위한 처리기 시간을 주기 태스크로 부터 스틸링 하는 방식으로 메모리 시스템에서의 사이클 스틸링과 개념적으로 유사한 방법이다. 슬랙 스틸링 알고리즘은 기본적으로 다음과 같은 가정을 하고 있다.

・문맥교환(context switching)이나 태스크 스케쥴링을 위한 부담(overhead)은 무시될 수 있을 만큼 작다.

・태스크의 준비시간은 주기의 시작과 일치하며, 다른 태스크 와의 동기나 자체적인 suspend는 발생하지 않는다.

・모든 태스크는 언제든지 선점(preemption)이 가능하다.

・비주기 태스크를 위한 버퍼의 크기는 무제한이다.

슬랙 스틸링 알고리즘은 비주기 태스크를 수행시키기 위한 처리기 시간을 할당하기 위하여 다음과 같은 분석을 근거로 구성된다. 우선순위 i의 태스크를 기준으로 시간0로부터 임의의 시간 t까지의 전체 부하는 다음과 같은 식으로 나타낼 수 있다.

$W_i(t)=a_i(t)-Pi(t)+I_i(t)$

위의 식에서 $W_i(t)$는 구간 [0, t]에서의 전체 계산량을 나타낸다. $a_i(t)$는 비주기 태스크의 계산량이고, $P_i(t)$는 우선순위 i의 태스크와 우선순위가 i이상인 주기 태스크의 계산 기간으로 다음과 같이 정의된다.

정의 1 $P_i(t)$는 t이전에 준비 시간이 경과한 우선순위 i이상의 모든 주기 태스크이 계산량이다. C를 태스크 τ_i의 최대 계산시간 이라고 하고, T_i를 주기라고 할때 $P_i(t)$는 다음과 같다.

$$P_i(t)(t) = \sum_{j=1}^{i} C_j \ \text{X} \ \frac{t}{T_j}$$

$I_i(t)$는 i-inactivity라고 정의되는 계산량으로 우선순위가 i보다 작은 주기 태스크의 계산량이다. $I_i(t)$는 우선순위가 i이상인 주기 태스크의 스케쥴링 가능성에 영향을 주지 않는 계산량이므로 비주기 태스크를 위한 계산 시간 할당에 이용된다.

$W_i(t)$의 스케쥴링 가능성은 그림 4.3의 Lehoczky의 정리 [Leh89]에 따라 다음과 같이 정의된다.

$$\min \ (0 \leqq t \leqq D_{ij}) \left\{ \frac{W_i(t)}{t} \right\} = 1$$

D_{ij}는 주기 태스크 τ_i의 j번째 인스탄스의 제한시간이다.

이때 $I_i(t)$는 τ_i의 스케쥴링 가능성에 영향을 주지 않으므로 위의 식은 다음과 같이 정리된다.

- $L_i(t) = \frac{P_i(t)}{t}$
- $L_i = min_{\{0<t\leq T_i\}} L_i(t)$
- $L = max_{\{1\leq i\leq n\}} L_i$

- **Lehoczky의 정리**: Given periodic tasks $\tau_1, \tau_2, ..., \tau_n$
  - τ_i can be scheduled for all task phasings using the rate monotonic algorithm if and onlt if $L_i \leq 1$
  - The entire task set can be scheduled for all task phasing using the rate monotonic algorithm if and only if $L \leq 1$

그림 4.3 : Lehoczky의 정리

	Period	Computation Time
τ_1	1	4
τ_2	3	6

표4.1: 주기 태스크의 예

$$\min\ (0 \leqq t \leqq D_{ij})\left\{\frac{A_{ij}+P_i(t)}{t}\right\} \leqq 1$$

위의 식에서 C_{ij}를 τ_1의 j번째 인스탄스의 종료시간이라 하면 A_{ij}는 구간 $[C_{ij-1}, C_{ij}]$에서 $A_i(t)$의 최대값으로 A_{ij}가 결정되면 $A_i(t)$는 다음과 같이 재정의 된다.

$$A_i(t) = A_{ij}, C_{ij} \leqq t \leqq C_{ij}, J \geqq 1$$

결국 A_{ij}는 구간$[C_{ij-1}, C_{ij}]$에서 우선순위 i로 비주기 태스크네 할당 가능한 처리기 시간이다. 이러한 분석을 근거로 표 4.1의 주기 태스크를 제한시간 이내에 스케쥴링 하면서 서비스 가능한 비주기 태스크의 계산시간을 구한 결과는 그림 4.4와 같다.

그림 4.4에서 $A_i(t)$는 우선순위 i에서 비주기 태스크에 서비스 가능한 계산시간이다. 그러므로 모든 주기 태스크가 제한 시간을 준수하면서 비주기 태스크에 할당 가능한 시간은 다음과 같이 정의되는 $A^*(t)$이다.

$$A^*(t) = \min_{1 \leqq j \leqq n} A_j(t)$$

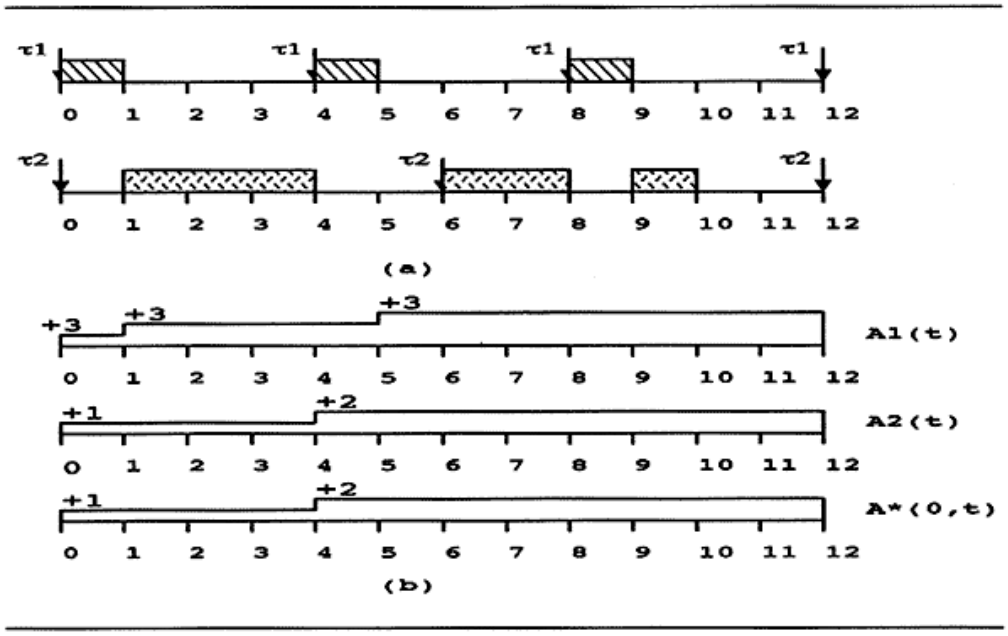


그림 4.4 : 슬랙 스틸링 알고리즘에서의 가용 슬랙 시간의 계산

이상의 결과는 시구간 $[0,t]$를 기준으로 비주기 태스크에 할당 가능한 시간인 $A^*(t)$를 구한 것으로 비주기 태스크가 시점 0에서부터 충분한 양이 도착되어 수행준비 상태일 때 사용가능하다. 그런데 비주기 태스크의 도착 시간은 예측 가능하지 못하므로 $[0,t]$ 사이의 임의의 시점 s에서 비주기 태스크에 할당 가능한 계산시간은 $A^*(t)$을 이용하여 알 수 없다. 그러므로 실질적인 이용을 위하여서는 $[s,t]$에서 사용 가능한 비주기 태스크 할당 가능 시간이 정의될 필요가 있다. 이 값을 $A^*(s,t)$라고 할때, $A^*(s,t)$는 $[s,t]$에서 우선순위 i에서 우선순위 i이상의 주기 태스크의 스케쥴링 가능성을 침해하지 않으면서 할당 가능한 시간인 $A_i(s,t)$로부터 구할 수 있다. 즉

$$A^*(s,t) = \min_{1 \leq j \leq n} A_j(s,t)$$

이다. $A_i(s,t)$는 다음과 같이 정의된다.

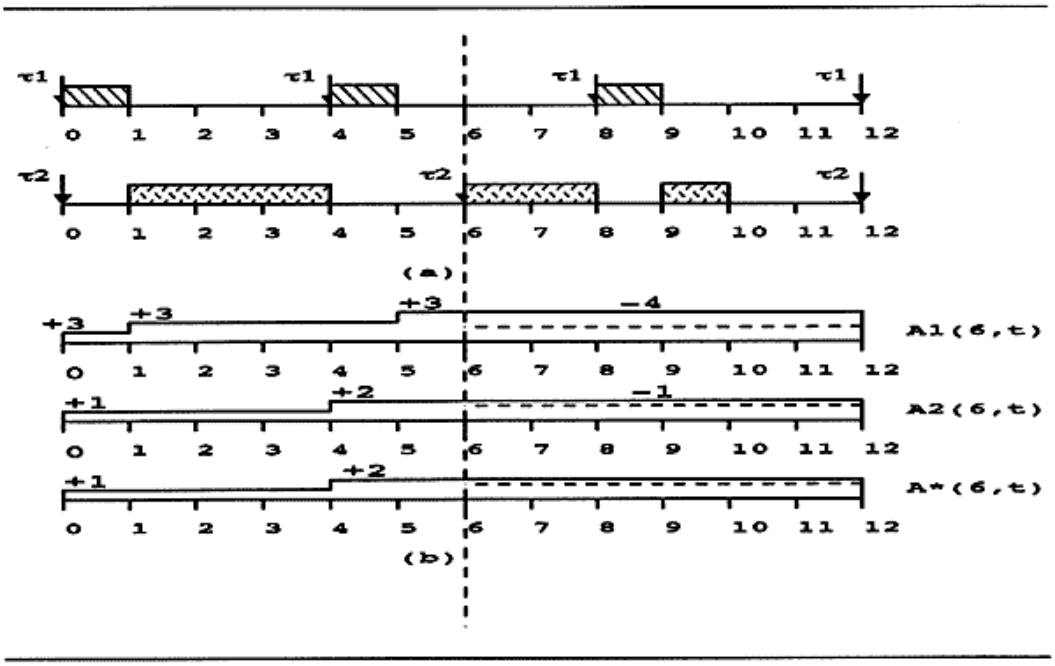


그림 4.5 : 임의 싯점에서의 슬랙 시간의 결정

$A_i(s,t) = A_{ij} - (A(s) + I_i(s)),\ C_{ij-1} \leqq t \leqq C_{ij}$

위의 식에서 $A(s)$는 $[0,s]$에서 수행된 비주기 태스크의 계산시간이고, $I_i(s)$는 $[0,s]$에서 idle시간을 포함한 level-i inactivity이다. A_{ij}에서 $A(s)+I_i(s)$를 제외한 값으로 $A_i(s,t)$가 정의되는 이유는 A_{ij}는 구간 $[0,t]$에서의 $A_i(t)$와 $I_i(t)$로 구성되는 고정적인 값이기 때문이다. 그림 4.5는 표 4.1의 주기 태스크에 대하여 구간 [0, 5.5]에서 비주기 태스크가 발생하지 않고 진행된 경우에 계산된 $A^*(5.5,t)$의 값을 나타낸 것이다.

슬랙 스틸링의 실질적인 구현은 다음과 같이 이루어진다.

· 모든 태스크 $\tau_i, 1 \leqq i \leqq n$에 대하여 $A_i(t)$, $0 \leqq t \leqq H$를 구한다. H는 모든 태스크의 주기의 최소 공배수로서 모든 태스크의 제한시간이 일치하는 시간이다. 계산된 A_i(t)의 값은 테이블로 저장된다.

· 실행 시간 서버는 태스크의 실행을 다음과 같이 분류한다.

- 0 : 비주기 태스크
- 1, ..., n : 우선순위 i, $1 \le i \le n$의 주기 태스크
- n + 1 : idle

· 위의 분류에 따라 다음과 같이 A와 I_i 의 값을 유지한다.

- 0 : add $t_2 - t_1$ to A
- 1 : do nothing
- $2 \le j \le n$: add $t_2 - t_1$ to $I1$, ..., I_{j-1}

· 임의의 시점 s에서 비주기 태스크를 위한 가용시간은 실행시간 서버에 의하여 다음과 같이 구한다.

$A^*(s, H) = min\ (A_i(H) - I_i - A)$

· $A^*(s, H)$ 의 값에 따라 비주기 태스크는 다음과 같이 처리된다.

- $A^*(s, t) \ge W$: 가장 높은 우선순위에서 즉시 처리
- $A^*(s, t) < W$: $[s, s + A^*(s, t)]$동안 $A^*(s, t) = A_i(s, t)$, $I \le i \le n$인 우선순위 i 로 처리

4.2.3 문제점 분석 및 문제 정의

이제까지 살펴본 봐와 같이 고정 우선순의를 가지는 주기 태스크와 동적으로 발생하는 비주기 태스크를 함께 스케쥴링 하는 혼합 태스크 스케쥴링 알고리즘으로는 폴링 서버 방식과 슬랙 스틸링 방식이 있다. 본 단에서는 이제까지 살펴본 기존의 스케쥴링 알고리즘의 문제점에 대하여 논한다.

폴링 서버 방식은 비주기 태스크를 위한 폴링 태스크를 주기적으로 동작 시켜서 비주기 태스크의 응답시간을 개선하였다. 그러나 폴링 서버 방식은 비주기 태스크가 항상 충분히 준비되어 있다는 가정 하에서 좋은 효과를 얻을 수 있는 방식으로 실질적으로는 비주기 태스크가 고른 분포로 발생하지 않을 수도 있으므로 비주기 태스크의 발생이 폴링 태스크의 주기와 일치하지 않으면서 발생할 수도 있으며, 이러한 경우 폴링 서버의 시간이 소모되거나, 그림 4.2에서와 같이 비주기 태스크의 서비스가 다음 주기의 폴링 태스크에 의하여 수행됨으로 인하여 대기시간이 길어져서, 응답시간의 최소화를 기대할 수 없다. 이러한 불필요한 대기시간이 발생하는 폴링 서버 방식의 단점을 개선한 알고리즘이 슬랙 스틸링 방식이다. 슬랙 스틸링 알고리즘은 비주기 태스크의 발생에 따라 슬랙 스틸링 서버가 적합한 우선순의를 비주기 태스크에 부여하여 즉시 서비스 가능하도록 함으로서 불필요한 대기 시간을 최소화하고 있다.

슬랙 스틸링 방식의 문제점은 RM 스케쥴링에 의해 할당된 주기 태스크의 우선순위가 동적으로 발생하는 비주기 태스크의 수행으로 인한 주기 태스크들간의 긴급도 변화에 대응하지 못한다는 것이다. 예로서 표 4.2의 주기 태스크를 스케쥴링 할 때, 1단위 시간의 최대 계산시간을 가지는 비주기 태스크가 시간 8에서 발생하였다 하면 슬랙 스틸링 알고리즘으로 스케쥴링 하는 경우에는 그림 4.6와 같이 T_3가 제한 시간을 넘기게 된다. 그러나 실제로는 1의 응답시간을 가질 수 있는 스케쥴링이 존재한다. 이러한 결과가 발생하는 이유는 그림 4.7과 같이 슬랙 타임을 구하는 함수의 한 요소로서 주기 태스크의 필요

	Period	Computation Time
T_1	2	6
T_2	2	8
T_3	3	12
T_4	2	24

표 4.2: 주기 태스크의 예

계산시간 값 $P_i(t)$가 긴급하지 않은 상위 우선 순의 태스크 A의 계산시간까지 포한하는 데에 있다.

이러한 문제점을 해결하여 비주기 태스크를 대기시간을 최소화하면서 스케쥴링 하기 위해서는 RM스케쥴링에 의한 스케쥴링 가능성을 위배하지 않으면서 동적으로 발생하는 긴급도의 변화에 적응 할 수 있는 우선순위 체계가 개발되어야 하며, 이러한 우선순위 체계하에서 비주기 태스크를 위한 계산기산을 할당할 수 있는 서버의 개발이 필요하다.

4.3 비주기 태스크의 응답 시간 개선을 위한 이중 우선순위 알고리즘

본 절에서는 슬랙 스틸링 알고리즘을 개선하기 위한 이중 우선순위 알고리즘을 제한하고, 제안된 고리즘이 RM 알고리즘과 최소한 같은 스케쥴링 가능성을 가짐을 보인다. 또한 제안된 알고리즘을 이용하여 주기 태스크에 우선순위를 할당한 이후, 슬랙 스틸링 알고리즘을 사용할 때, 비주기 태스크의 응답시간에 개선이 있음을 보인다.

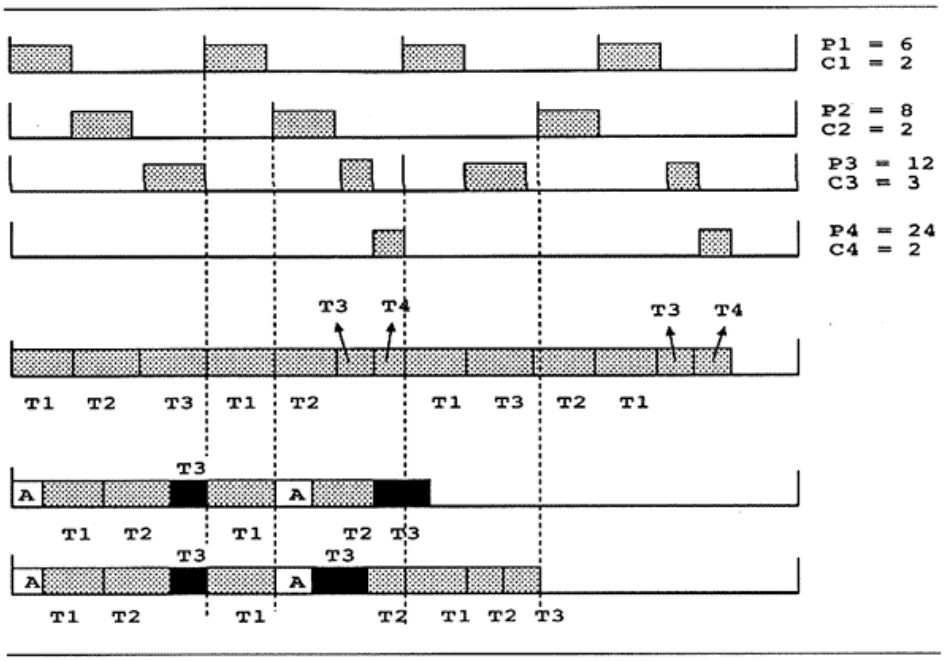


그림 4.6: 슬랙 스틸링의 단점

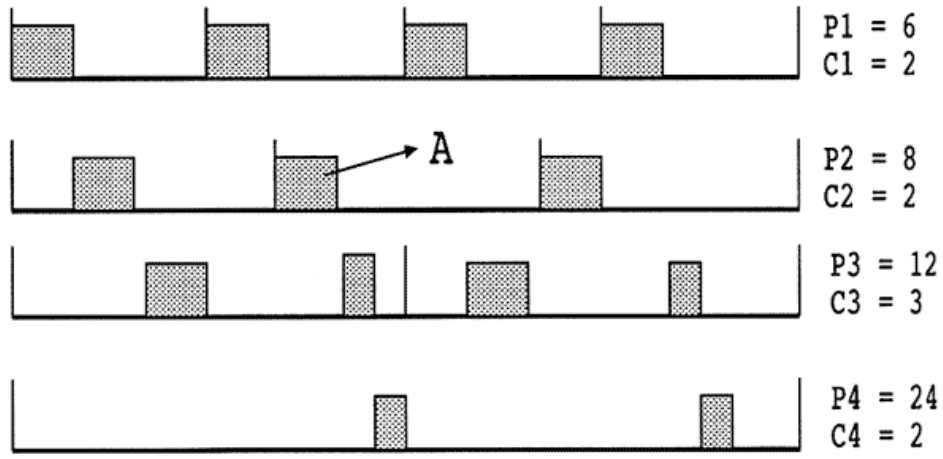


그림 4.7: Lehoczky 분석의 단점

4.3.1 혼합 태스크 스케쥴링의 환경 및 가정

혼합 태스크 스케쥴링의 환경은 그림 4.8와 같이 정적 스케쥴링 부분과 동적 스케쥴링 부분으로 구성된다. 정적 스케쥴링 부분에서는 주기 태스크에 고정 우선순위를 부여하고 동적 스케쥴링에 필요한 자료를 생성한다. 동적 스케쥴링 부분에서는 처리기의 수행상태를 감시하면서 주기 태스크와 비주기 태스크를 정적 스케쥴링에서 할당된 우선순위와 자료에 따라 스케쥴링 하게 된다. 본 논문에서 제안된 이중 우선순위 알고리즘은 정적 스케쥴링 부분에 이용된다.

슬랙 스틸링 알고리즘에서 동적 스케쥴러의 역할을 하는 것은 슬랙 스틸링 서버이다. 슬랙 스틸링 서버가 얻은 수 있는 정보 중 별도의 오버 헤드가 없이 얻을 수 있는 것으로는 아래와 같은 것들이 있다. 아래의 정보들은 디스페처도 별도의 오버헤드 없이 이용가능하다.

- 주기 태스크의 우선순위

- 각 태스크가 처리기에서 수행된 시간

- 처리기의 수행상태 (태스크의 수행 혹은 idle)

- 처리기 선점 시도의 성공 여부

본 연구에서는 다음과 같이 스케쥴링 환경에 대한 가정을 한다.

- 단일 처리기 시스템

- 태스크들 간의 처리기 선점은 항상 가능하다.

- 태스크는 자체적으로 suspend 되지 않는다.

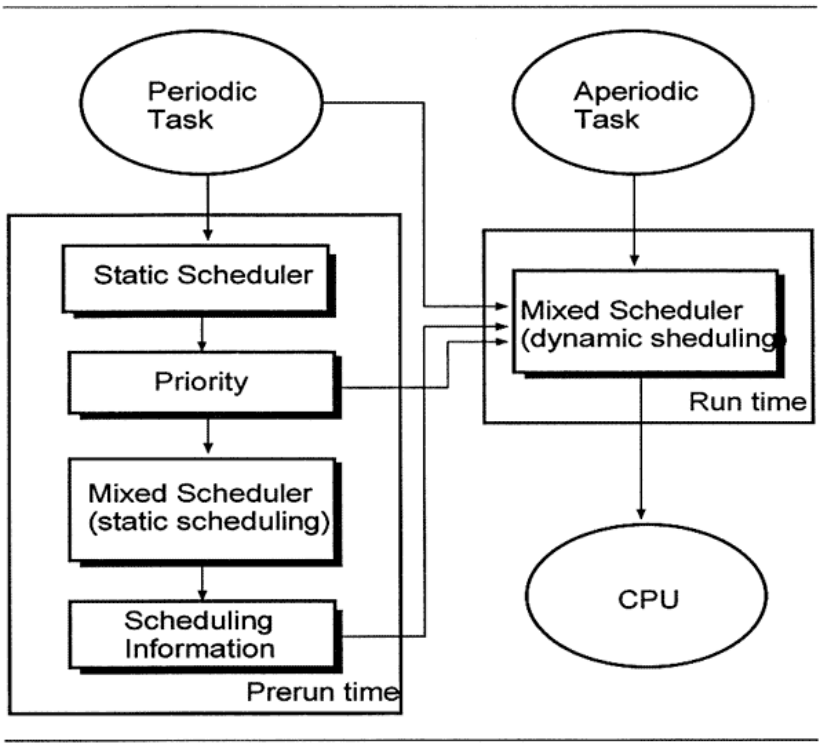


그림 4.8 : 혼합 스케쥴러의 구조

・context switching에 의한 오버헤드는 고려되지 않는다.

・주기 태스크는 엄격한 실시간 태스크이다.

・비주기 태스크는 연성 실시간 태스크이다.

・주기 태스크는 준비시간과 제한시간은 주기와 일치한다.

위의 가정 중에서 context switching에 의한 오버헤드의 무시는 실질적이지 못한 가정이기는 하지만 태스크의 계산시간에 포함을 시켜 설계한 후에 사용되지 않은 계신시간은 reclaimer등의 실행시간 서버를 이용하여 처리될 수 있으므로 알고리즘의 설계에서는 context switching의 오버헤드는 고려하지 않도록 한다. 또한 비주기 태스크가 엄격한 실시간 태스크인 경우에도 현재 시점에서 가용한 최대 시간을 얻어 낸다는 맥락에서 연성 실시간 시스템과 일치하므로 본 논문에서는 비주기 태스크를 연성 실시간 태스크로 제한 한다.

4.3.2 기본적 발상

본 연구의 기본적인 발상은 비주기 태스크를 위한 계산시간의 할당을 최대화하기 위해서, 슬랙 타임이 부족한 태스크는 그림 4.9와 같이 상위 우선순위의 태스크로부터 다시 슬랙 타임을 얻어 오는 것이다. 그림 4.9에서 화살표는 RM 스케쥴링과 다른 형태로 디스펫처가 동작한 부분으로, 만약 RM 우선순위에 따라 스케쥴링 하면 그림 4.10과 같이 제한 시간을 넘기게 된다. 그림 4.9와 같은 스케쥴링이 가능하기 위해서는 주기 태스크의 수행도중 일정한 시점에서 자신보다 상위의 RM우선순위를 가지는 태스크들의 처리기 선점을 막을 수 있는 방법이 제공되어야 한다. 이러한 처리기 선점의 방지책으로서 태스크 수행도중의 우선순위 변환에 의한 방법이 있을 수 있다.

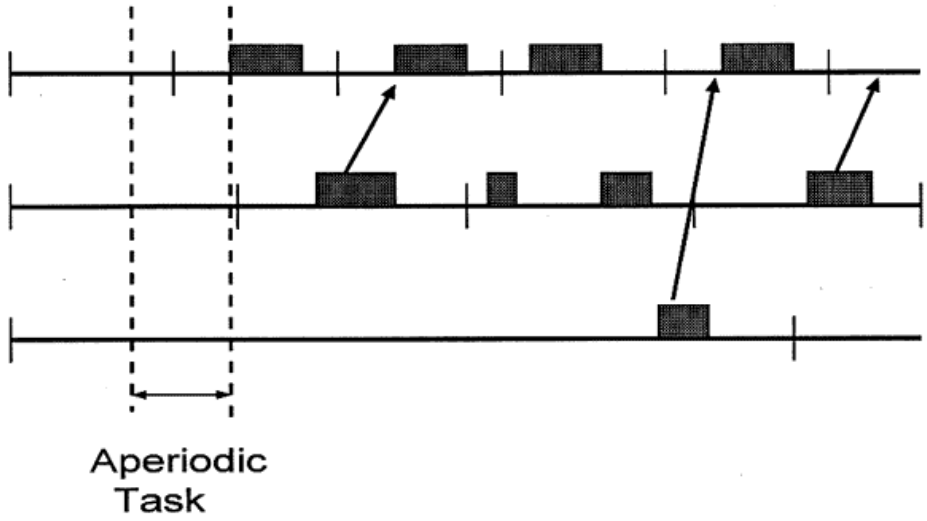


그림 4.9: 낮은 우선순위 태스크에 의한 상위 우선순위 태스크의 슬랙 조정

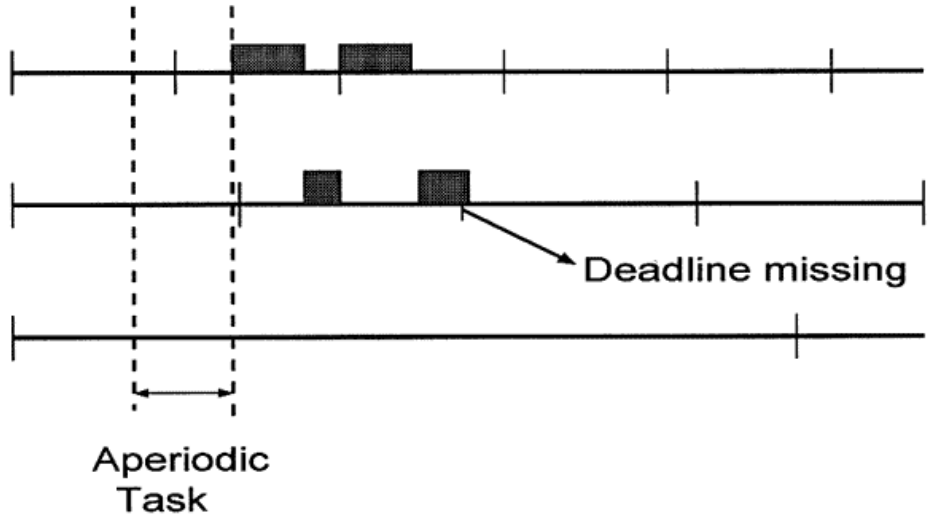


그림 4.10: RM 스케쥴링에 의한 결과

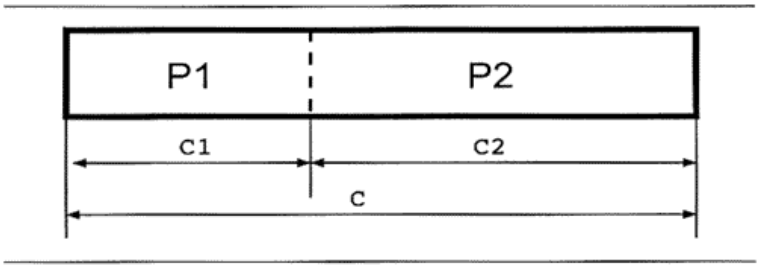


그림 4.11 : 태스크의 분할

RM우선순위가 상위인 태스크들로 부터의 선점을 방지하기 위한 우선순위의 변화는 한번이면 충분하다. 그러므로 태스크의 형태는 그림 4.11과 같이 2부분의 나뉘어져서 별도의 우선순위를 할당 받아야 한다. 이때, 태스크에 할당되는 우선순위는 RM 스케쥴링의 스케쥴링 가능성을 위배하지 않는 우선순위 할당이어야 하며, 뒷부분의 우선순위는 RM 우선순위 상에서 자신보다 높은 우선순위를 가지는 모든 태스크로부터 선점당하지 않을 수 있어야 한다. 이렇게 상호 독립적인 우선순위의 할당이 실행 시간 동안에 줄 수 있는 오버헤드는 상기한 바와 같이 시간 정보는 별도의 오버헤드 없이 얻을 수 있으므로 디스페쳐에서 한단계의 선택 문을 더 수행하는 것 이외에는 없다. 디스펫쳐의 구성비교는 그림 4.12와 같다. 디스펫쳐의 시행 횟수도 단일 우선순위 할당 때와 동일하다.

4.3.3 이중 우선순위 알고리즘

본 단에서는 이중 우선순위 알고리즘을 제안한다. 이중 우선순위 알고리즘의 목적은 다음의 2가지 사항을 만족하는 주기 태스크의 고정 우선순위 할당에 목적이 있다.

・RM 스케쥴링에 의해 스케쥴링 가능한 태스크는 새로운 우선순위 할당에 의해서도 스케쥴링이 가능하다.

Traditional Dispatcher	New Dispatcher
1. insert new task into task Q	1. insert new task into Q
2. select minimum priority task in Q	2. select minimum priority task in Q

3. if idle then execute selected task
else {
if current task priority >= selected task then
execute current task
else
execute selected task
}
	3. get used processor time

	4. if idle then execute selected task
	else {
	get executed time of current task
	if executed time >= Phase 1 time
	then current task priority = Phase 2 priority
	
	if current task priority >= selected task then
	execute current task
	else
	execute selected task

그림 4.12 : 디스펫쳐의 비교

· RM 스케쥴링으로 주기 태스크를 스케쥴링할 때보다 최소한 같거나 보다 적은 대기시간을 제공하면서 비주기 태스크를 수행시킬 수 있어야 한다.

우선순위의 할당방법을 설명하기 위하여 다음과 같이 몇 가지 정의를 한다.

정의2 태스크 τ_i가 분할된 경우 앞부분은 $J_{i,1}$이라 하고 뒷부분은 $J_{i,2}$라 한다.

정의3 $J_{i,1}$의 계산시간은 $c_{i,1}$이라 하고 $J_{i,2}$의 계산시간은 $c_{i,2}$라 한다. 태스크 τ_i의 계산시간이 c_i인 경우

$$c_i = c_{i,1} + c_{i,2}$$

이다.

정의 4 $J_{i,1}$의 우선순위는 $p_{i,1}$이며 $J_{i,2}$의 우선순위는 $p_{i,2}$이다.

정의 5 태스크 τ_i의 주기는 T_i이다.

본 연구에서 제안하는 우선순위 할당은 주기 태스크 $\tau_i,\ldots,\tau_n$이 주어지고, 이때 $T_i \le T_j$이면 $i \le j$이라 할 때 다음과 같다.

・τ_1은 분할되지 않는다.

・if $i \le j$ then $p_{i,1} \ge p_{i,1}$, $1 \le i \le n$

・$p_{i,2} > p_{i,2}$, $2 \le j \le i-1$

・$p_{i,2} < p_{i,2}$, $2 \le i \le n$

위의 조건을 만족하는 우선순위 할당을 할 경우, 그림 4.13에서 화살표가 진행하는 방향에 따라 상대적으로 낮은 우선순위가 부여되게 된다.

이러한 우선순위의 할당이 RM 스케쥴링에 의해 스케쥴링 가능한 태스크를 스케쥴링 할 수 있기 위해서는 $c_{i,2}$의 적절한 할당이 필요하다. $c_{i,2}$의 결정방법을 설명하기 위하여 몇 가지 정의를 내리도록 한다.

정의6 Ri는 τ_i가 RM 스케쥴링에 의하여 스케쥴링 될 경우의 최대 응답시간이다.

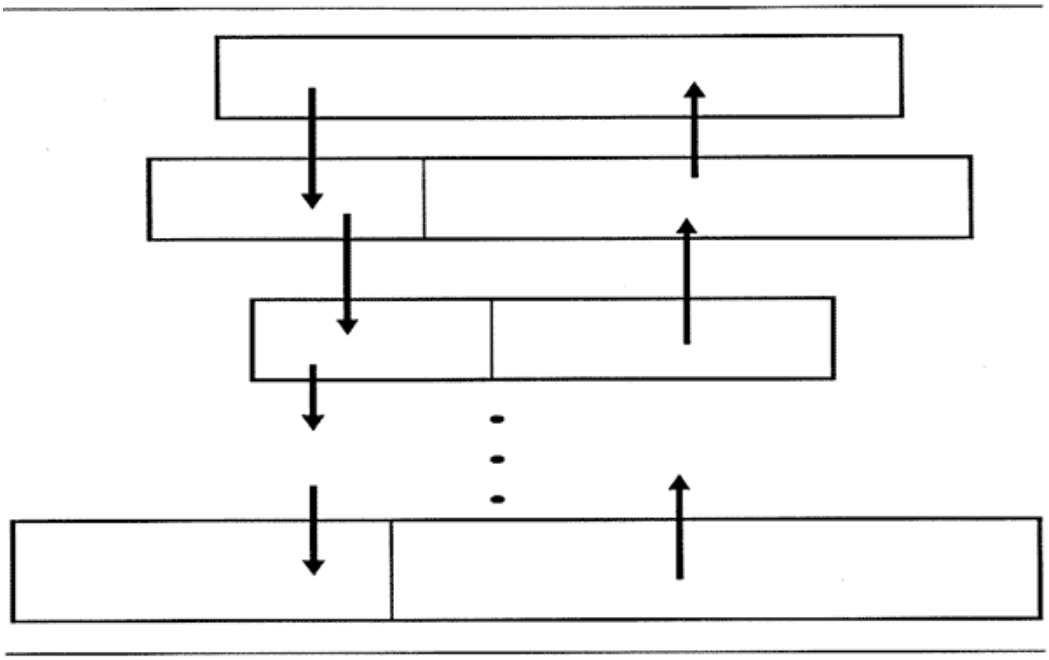


그림 4.13 : 우선순위의 할당

$$R_i = c_i + \sum_{j=1}^{i-1} [\frac{T_i}{T_i}] \qquad C_j$$

정의 7 각 태스크의 슬랙 시간 S_i는 다음과 같이 정의한다.

$S_i = T_i - R_i$

	Computation Time	Period
T_1	5	20
T_2	5	25
T_3	20	65
T_4	10	110

표 4.3:주기 태스크의 예

정의 8 최소 슬랙 시간 S_M는 다음과 같다.

$S_M = min_{1 \le i \le n} S_i$

$c_{i,2}$, $2 \le i \le n$의 결정 방법은 다음과 같다.

$si = min \le j \le i-1\{max\{0, Tj-Rj-SM\}\}$

$c_{i,2} = min\{c_i - \epsilon, s_i\}$

위의 식에서 ϵ은 최소 계산시간 값으로 실제 구현에서는 context switching에 의한 오버헤드에 대하여 손실이 없는 분기점의 계산시간이다. 예로서 4.3의 주기 태스크에 우선순위를 할당했을 때의 결과는 표 4.4와 같이 나타난다.

	Slack S_i	Minimun Slack S_M	$c_{i,1}$	$c_{i,2}$	$p_{i,1}$	$p_{i,2}$
T_1	15	5	5		4	
T_2	10	5	1	4	5	3
T_3	20	5	15	5	6	2
T_4	50	5	5	5	7	1

표 4.4: 이중 우선순위 할당의 결과

4.3.4 이중 우선순위 할당의 정당성

본 단에서는 앞서 서술한 이중 우선순위 알고리즘이 RM 알고리즘과 최소한 같은 스케쥴링 가능성을 가짐을 보임으로서, 이중 우선순위 알고리즘이 주기 태스크 스케쥴링에서 RM알고리즘을 대신할 수 있음을 입증하고자 한다.

정의 9 이중 우선순위 스케쥴링에서 임의의 태스크 $\tau_i, 1 \le i \le n$가 $T_i \ge T_j$, $j \ne i$, $1 \le j \le n$인 태스크 τ_j들로부터 자신의 주기가 시작된 이후부터 종료 시간사이에 선점 당할 수 있는 최대 시간량을 H_i'이라 하고, RM 스케쥴링에서 동일한 조건의 시간량을 H_i이라 하고, RM스케쥴링에서 동일한 조건의 시간량을 H_i라 한다.

정의 10 이중 우선순위 스케쥴링 에서 임의의 태스크 $\tau_i, 1 \le i \le n$가 $T_i \le T_j$, $j \ne i$, $1 \le j \le n$인 태스크 τ_j들로부터 자신의 주기가 시작된 이후부터 종료 시간사이에 선점 당할 수 있는 최대 시간량을 B_i라 하고, 이러한 현상을 블록킹(blocking)이라 한다.

정의 11 임의의 주기 태스크 집합 *T*가 *RM*스케쥴링으로 스케쥴링 가능하면 T를 *RM feasible*하다고 한다.

보조 정리1 *RM-feasible*한 태스크 집합 $T = \{\tau_1,...,\tau_n\}$가 주어질 때, 항상 다음의 조건이 만족된다.

$H_i \le H_i'$, $1 \le i \le n$

증명:
RM-feasible한 임의의 태스크 집합 $T = \{\tau_1,...,\tau_n\}$가 주어지고, T에 입증 우선순위가 정의되어 있다 하자. 태스크는 주기의 크기가 감소하지 않는 순서로 정열 되어 있다고 할 때, 임의의 태스크 $\tau_i \in T$, $1 \le i \le n$가 $k \times T_i$시간에 준비되었다 하면,

$p_{i,1} \le p_{j,k}$, $k = 1,2$; $1 \le j \le i-1$

이므로 $(k+1) \times T_i$이전에 준비되는 태스크 τ_j, $1 \le j \le i-1$들이 τ_i가 $c_{i,1}$만큼 수행되기 이전에 도착하는 경우에는 $H_i = H_i'$이다. 그러나 그렇지 않은 경우에는

$p_{i,2} \ge p_{j,k}$, $k=1,2$; $1 \le j \le i-1$

이므로

$H_i - \epsilon = H_i'$, $\epsilon > 0$

이 된다.

그러므로

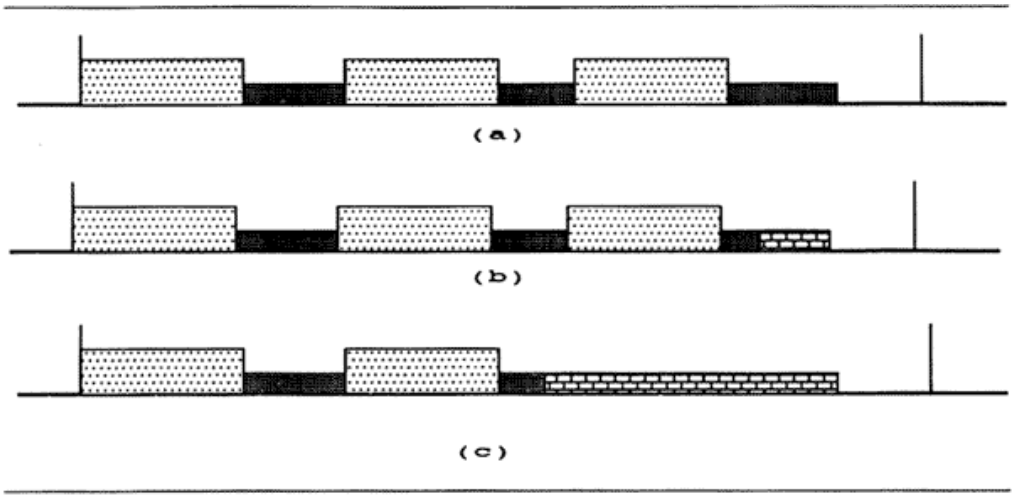


그림 4.14 : 보조 정리 1의 예

$H_i = H_i'$, $1 \le i \le n$

이다.

보조정리 2 태스크 집합 T = $\{\tau_1,\ldots,\tau_n\}$에 이중 우선순위가 정의될 때, τ_1, $1 \le i \le n$가 $T_i \le T_j$인 태스크 τ_j로 인하여 구간 $[k \times T_i, (k+1) \times T_i]$안에서 받을 수 있는 블로킹은 최대 1번이다.

증명 : 임의의 주기 태스크 τi보다 주기가 큰 태스크들을 τ_k라 하자. 태스크 $\tau_i \in T$,가 처리기를 선점하는 경우, pi,1보다 낮은 우선순위의 태스크가 현재 수행중인 상황이므로, τ_i가 종료한 이후에는 $J_{k,1}$부분을 수행하게 된다. 또한 τ_i가 다른 태스크를 블로킹하는 경우 τ_i가 $J_{i,1}$을 종료하여야 하므로 τ_i보다 주기가 작은 태스크들은 τ_i가 $J_{i,1}$ 을 수행하는 동안 이미 종료 상태이거나 준비되지 않은 상태이다.

결국 τ_i의 종료 이후 준비된 태스크들은 모두 $J_{k,1}$에 있으므로 $J_{i,2}$의 수행은 발생되지 않는다. 그러므로 하나의 태스크 인스탄스가 브로킹 당하는 횟수는 1번이하이다.

정의 12 이중 순위 체계에서 각 태스크의 최대 응답 시간은 R_i'이라 정의 한다.

보조 정리3

$R_i' \leq T_i$

증명 : 보조 정리 1과 2에 따라 이중 우선순위 하에서 태스크 τ_i의 최대 종료시간은

$R_i' = B_i + H_i' + c_i$

이다. 그런데 $H_i' \leq H_i$이므로

$R_i' \leq B_i + H_i' + c_i$

이다. 이때 $B_i = \min_{1 \leq j \leq i}(S_i - S_M)$이므로, $B_i < S_i$이다. 그러므로

$B_i \leq B_i + H_i + ci < S_i + H_i + c_i$ 이다.

$S_i = T_i - (H_i - ci)$이므로 위의 부등식은 다음과 같이 정리된다.

$R_i < T_i$

정리 1 태스크 집합 T가 RM 스케쥴링이 가능하면 이중 우선순위 할당에 의하여서도 스케쥴링이 가능하다.

증명 RM-feasible한 태스크 집합 T에 대하여 보조정리 1, 2, 3에 의해 태스크의 최대 종료시간 R_i가 항상 주기이내에 있으므로 이중 우선순위 알고리즘도 T를 스케쥴링 가능하다.

4.3.5 슬랙 스틸링의 개선

본 단에서는 앞서 제한한 이중 우선순위 알고리즘에 의하여 할당된 이중 알고리즘으로 주기 태스크를 수행하면서 슬랙 스틸링을 하는 예를 보이고, 비주기 태스크의 응답 시간 개선을 입증한다.

이중 우선순위를 이용하여 슬랙 스틸링을 할 때는 2장에서 자세히 설명한 슬랙 스틸링을 위한 함수 중 몇 가지를 수정하여야 한다. 수정된 함수는 다음과 같다.

$$min_{0 \le t \le Dij - Ci,j} \frac{A_{ij} + P_i(t)}{t} = 1$$

$A_i(t) = A_{ij},\ C_{i,j-1} - c_{i2} \le t \le C_{ij} - c_{i2}$

또한 실행시간에 기록되는 각 태스크의 수행 기록은 관리할 때에도 τ_i 의 우선순위에 대한 기록은 $p_{i,1}$으로 하며, 블러킹이 발생할 때에만 블러킹한 시간만큼을 pi,2로 기록한다.

이렇게 변형된 슬랙 스케쥴링 알고리즘으로 스케쥴링한 예는 그림 4.15와 같다. 그림에서 괄호안의 수는 나뉘어진 계산시간이다. 예에서는 τ_3이 가장 슬랙 타임 작아서 τ_4의 분할은 이루어지지 않았다. 예에서 시간 6에 2의 계산 시간을 가지는 태스크가 도착하여 수행될 수 있었다. 그러나 그림의 예에서 가장 밑에 나타난 것처럼 RM에 의한 슬랙 스틸링을 하는 경우에는 τ_3이 제한 시간을 넘기게 된다.

이제 RM 우선순위를 이용할 경우보다 이중 우선순위를 이용했을 때의 슬랙 스틸링 알고리즘에 의한 비주기 태스크의 응답시간이 작음을 보인다.

정의 13 *RM* 우선순위에서 $A_i(t)=A^*(t)$인 τ_i를 *critical* 태스크라고 한다.

정의 14 *RM* 스케쥴링에서 τ_i의 최대 종료 시감을 Ci라 한다. 또한 이중 우선순위 스케쥴링에서 τ_i의 최대 종료시간을 C_i'라 한다. 또한 이중 우선순위 스케쥴링애서 τ_i의 최대 종료시간을 C_i'라 한다.

보조 정리 4 *critical task* τ_i에 대하여 항상 $C_i' \leq C_i$이다.

증명 : 이중 우선순위가 태스크 집합 T에 할당될 때, 태스크 τ_i의 최대 종료사건 C_i'는 다음과 같다.

$C_i' = B_i + H' + c_i$

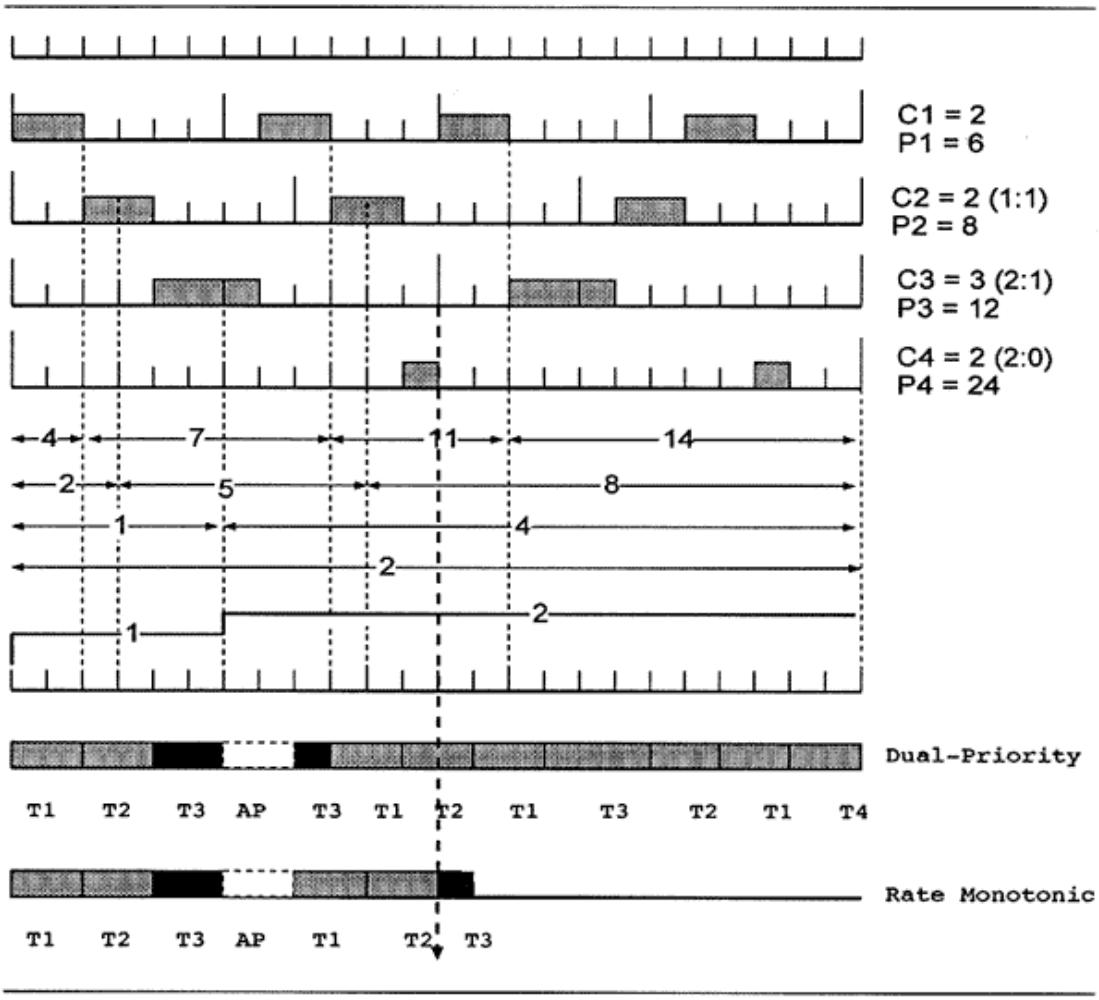


그림 4.15 : 개선된 슬랙 스틸링의 수행

그런데 $S_m = S_i$이므로 $B_i=0$다. 그러므로 위의 식은 다음과 같이 변환될 수 있다.

$C_i' = H_i' + c_i$

이때 $C_i = Hi + ci$이고, $H_i' \leq H_i$이므로

$C_i' \leq C_i$

이다.

정리 2 *RM*스케쥴링에 의한 슬랙 스틸링의 결과를 $A^*_{RM}(t)$라하고 이중 우선순위에 의한 슬랙 스틸링의 결과를 $A^*_{DP}(t)$라 할 때

$A^*_{RM}(t) \leq A^*_{DP}(t)$

이다.

증명: $A^*(t)$는 항상 critical task에 의하여 영향을 받는다. 그런데 $A_i(t)$의 값은 태스크의 종료시간에 증가하며, $C_i' \leq C_i$이므로 임의의 시점 t에서 항상

$A^*_{RM}(t) \leq A^*_{DP}(t)$

임이 명백하다.

	Computation Time	Period
T_1	4	20
T_2	5	65
T_3	20	150
T_4	30	320
T_5	30	430
T_6	60	500

표 4.5: 주기의 차이가 큰 주기 태스크 집합

그러므로 이중 우선순위에 의한 슬랙 스틸링은 항상 RM에 의한 슬랙 스틸링 보다 많은 계산시간을 할당함으로 비주기 태스크의 응답시간을 줄일 수 있다.

4.4 실험 및 평가

본 절에서는 이중 우선순위에 의한 슬랙 스틸링이 RM 우선순위에 의한 슬랙 스틸링 보다 실제로 얼마만큼의 성능 향상이 있는지를 알아보기 위한 간단한 실험 결과를 보이고 이에 대해 논한다.

4.4.1 주기의 형태에 의한 영향 실험

본 단에서는 주기 태스크가 갖는 주기의 형태에 의한 영향을 실험을 통하여 알아본다. 이중 우선순위 할당의 기본적인 개념은 서로 교차하는 주기의 형태를 이용하여 슬랙 시간

표 4.6: 주기의 차이가 작은 태스크의 집합

	Computation Time	Period
T_1	4	20
T_2	3	30
T_3	+ 2	45
T_4	6	50
T_5	8	120
T_6	10	180
T_7	15	210
T_8	10	320

의 효율적인 이용을 시도하는 것이므로 주기의 형태에 많은 영향을 받는다.

실험에 사용된 주기 태스크 집합은 2종류로 표 4.5와 표 4.6에 나타나 있다. 표 4.5는 인접한 태스크간의 주기가 크게 차이가 나는 태스크 이고, 표 4.6은 차이가 작은 태스크 집합이다. 비주기 태스크의 크기는 10 단위 시간 이하의 태스크들만이 발생하도록 조정 하였다. 비주기 태스크의 부하는 주기 태스크의 부하에 따라 전체 부하가 95%를 넘지 않도록 조정 하였으며, 비주기 태스크의 계산 시간은 10 단위 시간을 초과하지 않도록 하였다.

실험 결과에 의하면 주기 태스크들의 주기차가 적은 시스템일수록 이중 우선순위를 이용한 슬랙 스틸링이 좋은 것으로 나타나 있다. 또한 비주기 태스크의 부하가 커질 수록 이중 우선순위 알고리즘에 의한 슬랙 스틸링이 좋은 효과를 가지는 것으로 나타났다. 주기차가 작은 시스템에서는 비주기 태스크의 부하가 심한 경우 42%정도의 감소효과가 있었다. 주기의 차이가 큰 태스크 집합에서도 비주기 태스크의 부하가 큰 경우에는 18%의 응답 시간 감소효과가 있었다. 실험 결과가 상기한 바와 같이 나온 이유는 주기의 차가 크지 않아 태스크들의 주기가 서로 적절히 엇갈릴 경우, 이중 우선순위 알고리즘은 주기 태스크간에 슬랙을 잘 조절할 수 있기 때문이다.

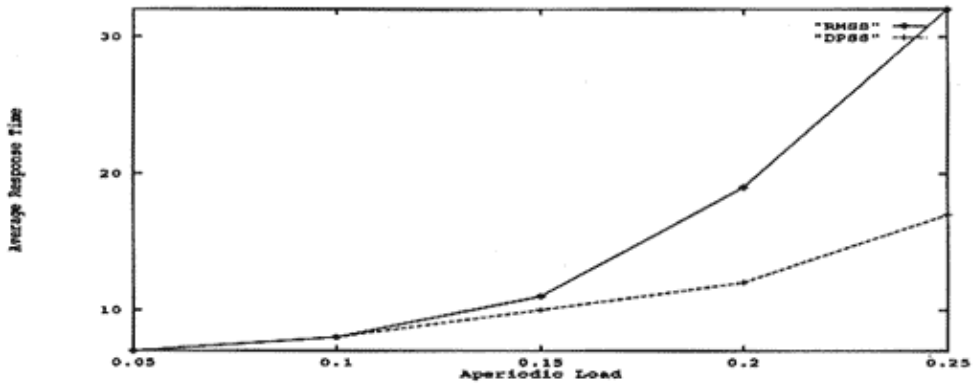


그림 4.16: 주기 차이가 작은 태스크 집합

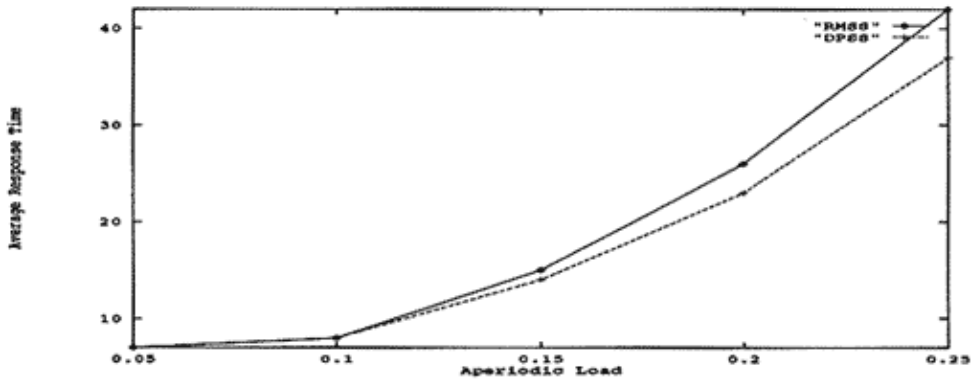


그림 4.17: 주기 차이가 큰 태스크 집합

	Computation Time	Period	Description
T_1	1	3	Update ship attitude
T_2	4	40	Update displacement
T_3	10	63	Send atitute message
T_4	20	1000	Send navigation message
T_5	100	1000	Update status on screen
T_6	25	1250	Update ship position

표 4.7:INS 주기 태스크

4.4.2 고유한 우선순위를 가지는 비주기 태스크의 실험

TDX-10과 같은 전전자 교환기 시스템이나, INS(Inertial Navigation System)에서는 비주기 태스크간의 우선순위가 비주기 발생시키는 서버들에 의하여 결정된다. 본 단에서는 이러한 경우의 실제 동작을 살펴보도록 한다.

실험에 사용된 주기 태스크는[LS92]에서 사용된 INS 시스템의 주기 태스크를 이용 하였다. 주기 태스크의 부하는 78%이며, 비주기 태스크의 부하는 20%이하로 조정하였다. 비주가 태스크의 계산 시간은 한 태스크가 25%를 넘지 못하도록 제한하였다. 비주기 태스크간의 우선 순위는 10단계로 구분하였으며, 비주기 태스크가 발생하여 태스크 큐에 저장될 때, 이 우선순위에 의하여 정렬되도록 하였다.

INS 태스크에서의 실험에서는 비주기 태스크의 부하가 심할 경우에는 응답 시간의 개선이 크게 나타나고 있지 않다. 이는 주기가 가장 작은 태스크와 다른 태스크간에 주기의 차가 너무 크기 때문에 발생하는 결과이며, 비주기 태스크의 부하가 비교적 작은 경우(15%)에는 10에서 15% 정도의 응답 시간 감소효과가 있었다. 그러나 비주기 태스크의 부하가 너무 작은 경우에는 이중 우선순위에 의한 효과를 거의 볼 수 없었으며, 거의 동일하거나 10% 미만의 개선이 있었을 뿐이다.

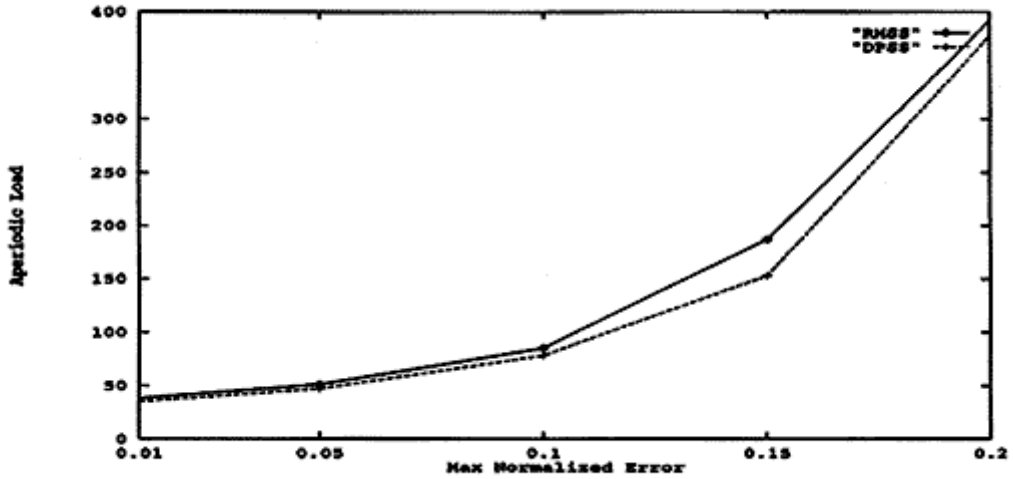


그림 4.18 : INS에서의 실험결과

이러한 결과는 INS 태스크의 특이한 주기 형태에 의한 것으로 비주기 태스크간의 고유 우선순위에 의한 영향은 없었다.

4.5 결 론

본 연구에서는 주기 태스크에 고정 우선순위를 할당하는 이중 우선순위 알고리즘을 제안하고, 이를 이용하여 슬랙 스틸링 서버 알고리즘을 개선하였다. 제안된 이중 우선순위 알고리즘은 슬랙 스틸링 서버 알고리즘이 비주기 태스크를 위한 슬랙 타임을 보다 많이 얻을 수 있도록 주기 태스크를 수행시킬 수 있는 고정 우선순위를 할당하는 알고리즘이다.

본 연구에서는 제안된 이중 우선순위 알고리즘이 RM 알고리즘과 최소한 동일한 스케쥴링 가능성을 가짐을 보임으로써, 슬랙 스틸링 서버 알고리즘에 이중 우선순위 알고리즘이 RM 알고리즘 대신 사용될 수 있는 정당성을 보였으며, 이때 RM 알고리즘으로 주기 태스크를 스케쥴링 하였을 때보다 많은 슬랙 시간을 얻을 수 있음을 보여 비주기 태스크의 응답시간이 개선됨을 입증하였다.

또한, 주기 태스크의 스케쥴 가능성에 큰 영향을 주는 주기의 형태에 따른 실험을 통하여 실제의 성능 개선과 주기의 형태에 따른 성능 개선의 차이를 알아보았다.

본 연구에서 제안된 알고리즘은 2장과 3장에서 조사 분석된 차세대 전전자 교환기를 위한 운영체제의 스케쥴링 알고리즘 요구사항을 고려하여 설계된 알고리즘으로 주기적인 태스크들의 수행 주기를 만족시키며, 비주기적인 태스크와 동적으로 발생하는 복구처리 프로세스의 응답시간을 최소화하는 알고리즘이다.

제 5 장 결론 및 앞으로의 연구 방향

5.1 결론

본 연구에서는 차세대 전 전자 교환기를 위한 실시간 운영체제의 스케쥴러 구현을 위하여 실시간 스케쥴링 알고리즘을 설계하였다.

광대역 통신망에서의 서비스는 멀티미디어 응용 등을 포함한 보다 폭넓고 서비스의 질이 높은 서비스가 요구될 것이다. 이러한 서비스의 지원을 위해서는 실시간 처리 기술의 지원이 필수적이며 이는 성능이 우수한 실시간 운영체제를 기반으로 지원될 수 있다. 실시간 스케쥴링 알고리즘은 이러한 실시간 운영체제 구현에 기반이 되는 기술이다.

본 연구에서 설계된 알고리즘은 고정 우선순위 체제에서 주기적인 태스크의 주기적 수행을 보장하면서, 비주기적으로 발생하는 태스크들의 응답시간을 최소화하는 스케쥴을 구성하는 알고리즘으로, 차세대 전전자 교환기의 실시간 운영체제뿐만 아니라, 멀티미디어 플랫폼의 스케쥴링, 엔진제어 시스템 및 각종 OBC 제어시스템에도 응용되어 이용될 수 있다.

5.2 앞으로의 연구 방향

앞으로의 연구 방향은 개발된 스케쥴링 알고리즘을 이용하여 보다 효율적으로 엄격한 실시간 제한을 가지는 비주기 태스크를 처리하는 방안의 연구 및 분산 시스템에서의 효율적인 전역 스케쥴링 알고리즘의 개발이다.

또한 이중 우선순위 알고리즘이 공유자원을 이용하는 경우의 영향을 분석하여 보다 실용적인 알고리즘으로 개선하는 연구이다.

참 고 문 헌

[Bla74] J. Blazewicz, "Scheduling dependent tasks with different arrival time to meet deadline," *Modelling and Performance Evaluation of Computer system*, 1974.

[CC89] H. Chetto and M. Chetto, "Some results of the earliest deadline scheduling algorithm," *IEEE Transactions on Software Engineering,* vol.15,pp. 1261-1269, Oct. 1989.

[CL88] J. Y. Chung and J. W. S. Liu, "Algorithms for scheduling periodic hobs to minimize average error," in *Proceeding of the 9th IEEE Real-Time Systems Symposium,* pp. 142-151, 1988.

[CPF75] K. M. Chandy and Reynold P. F, "Scheduling partially ordered tasks with probabilistic execution times," *The fifth Symposium on Operating Systems*, 1975.

[CW90] D. W. Craig and C. M. Woodside, "The rejection rate for tasks with random arrivals, deadlines, and preemptive scheduling," *IEEE Transactions on Software Engineering,* vol. 16, pp. 1198-1208, Oct. 1990.

[DL78] S. K. Dhall and C. L. Liu, "On a real-time scheduling problem," *Operations Research,* vol. 26, no. 1, pp. 127-140, 1978.

[DM89] M. L. Dertouzos and A. K. Mok, "Multiprocessor on-line scheduling of hard-real-time tasks," *IEEE Transactions on Software Engineering,* vol. 15, pp. 1497-1506, Dec. 1989.

[Efe82] K. Efe, "Heuristic models of task assignment scheduling in distributed systems," *IEEE Computer, Jun. 1982.*

[GJ77] T. Gonzalez and D. B. Johnson, "A new algorithm for preemptive scheduling of trees," *TR 222, Pensylvania Univ., 1977.*

[GKS91] A. Raghunathan G. Korean, B. Mishra and D. Shasha "Online schedulers for overload real-time systems," *Technical report no.558, Courant Institute, NYU,* May 1991.

[HL88] K. S. Hong and J. Y-T. Leung, "On-line scheduling of real-time tasks," in P*roceeding of the 9th IEEE Real-Time Systems Symposium,* pp.244-250, 1888.

[HL92] C. C. Han and K. J. Lin,"Scheduling distance constrainted realtime tasks," in *IEEE RTSS*, pp. 300-308, 1992.

[Hor74] W. A. Horn, "Some simple scheduling algorithms," *Naval Research Logistics Quarterly*. vol. 21, 1974.

[JWSLN87] K-L. Lin J. W. S. Liu and S. Nartarajan, "Scheduling real-time, periodic jobs using imprecise results," in *Proceeding of the 8th IEEE Real-Time Systems Symposiums, pp.* 252-260, 1987.

[KJLL87a] S. Natarajan K-J. Lin and J. W. S. Liu, "Concord: A distributed system making use of imprecise resurts," in *Proceeding of COMSPAC'* 87, 1987.

[KJLL87b] S. Natarajan K-J. Lin and J. W. S. Liu, "Imprecise results: Utilizing partial computations in real-time systems," in *Proceeding of the 8th IEEE Real-Time Systems Symposiums*, pp. 210-217, 1987.

[KS86] C. M. Krishna and K.G. Shin, "On scheduling tasks with a quick recovery from failure," *IEEE, 1986.*

[KS91] G. Koren and D. Shasha, "An optimal scheduling algorithm with a competitive factor real-time systems," T*echnical report no.558,*
Courant Institute, NYU, Jul. 1991.

[KS92] G. Koren and D. Shasha, "D-over: An Optimal on-line scheduling algorithm for overaoaded real-time systems," *Technical report no. 558,*
Courant Institute, NYU, Jan. 1992.

[Kis78] H. Kise, "A solvable case of the one machine scheduling problem with ready due times," *Operation researchs,* vol. 26, no. 1, 1978.

[Koo91] J. A. Stankovic, *Foundation of Real Time Computing.* Klewer Academic Publishers, 1991.

[Koo92] J. A. Stankovic, *Real-Time Systems Engineering and Applications*. Klewer Academic Publishers, 1992.

[LC86] A. L. Liestman and R. H. Campbell, "A fault tolerant scheduling problem," *IEEE TR on SE*, vol. 12, pp. 1089-1095, Now. 1986.

[LL73] C. L. Liu and J. W. Layland, "Scheduling algorithm for multiprogramming in a hard real-time environment," in *JACM*, vol. 20, pp. 46-61, 1973.

[LS92] J.P. Lehoczky and R-T. S. Ramos-Theul, "An optional algorithm for scheduling soft-aperiodic tasks in fixed priority preemptive system,"
in *IEEE RTSS*, pp. 110-123, 1992.

[LT92] J. P. Lehoczky and S.R. Theul, "An optimal algorithm for scheduling soft-aperiodic tasks in fixed priority preemptive systems,"
in *IEEE RTSS,* pp. 110-123, 1992.

[Law73] E. L. Lawler, "Optimal scheduling of a single machine subject to precedence constraints," *Management science,* vol. 19, 1973.

[Leh73] J. P. Lehoczky et. al., "Enhanced aperiodic scheduling in hard real time environment," in *IEEE RTSS*, pp. 261-270, 1987.

[Leh89] L. Sha J. Lehoczky and Y. Ding, "The rate monotonic scheduling algorithm: Exact characterization and average case behavior,"
in *IEEE RTSS,* pp. 166-171. 1989.

[Leh90] J. P. Lehoczky, "Fixed priority scheduling of periodic task sets with arbitrary deadlines," in *IEEE RTSS*, pp. 201-209, 1990.

[MD78] A. K. Mok and M. L. Dertouzos, "Multiprocessor scheduling in a hard real-time environment," in *Proceeding of the Seventh Texas Conference on Computing Systems*, Nov. 1978.

[MEC70] R. R. Muntz and E. G. Coffman JT., "Preemptive scheduling of real-time tasks on multiprocessor systems," *Journal of the As-sociation for Computing Machinery*, vol. 17, pp. 324-338, Apr. 1970.

[MJ70] R. R. Muntz and E. G. Coffman JT., "Preemptive scheduling of real-time tasks on multiprocessor systems," Journal of the As-sociation for Computing Machinery, vol. 17, pp. 324-338, Apr. 1970.

[Ram91] S. Ramos-Theul and J. K. Strosnider, "The transient server approach to scheduling time-critical recovery operation," in *IEEE RTSS*, pp.286-295, 1991.

[SL92] W. K. Shih and J. W. S. Liu, "On-line scheduling of imprecise computations to minimize error," in *Proceeding of the 13th IEEE Real-Time Systems Symposium,* pp. 280-289, Dec. 1992.

[SR88] J. A. Stankovic and K. Ramamritham, *Tutorial Hard Real-Time Systems*, IEEE Computer Society Press, 1988.

[Shi92] W. K. Shin, "Scheduling in real-time systems to ensure graceful degradation," *Technical Report, University of Illinois at Urbana-Champaign*, 1992.

[Spr89] L. Sha B. Sprunt and J. P. Lehoczky, "Aperiodic task scheduling for hard real time system," in Journal of *Real-Time Systems*, vol. 1, pp. 27-60, 1989.

[Ull76] J. D. Ullman, "Complexity of sequence problem," *Computer and Job Shop Scheduling*, 1976.

[WKSC89] J. W. S. Liu W. K. Shih and J. Y. Chung, "Fast algorithms for scheduling imprecise computations," in *Proceeding of the 10th IEEE Real-Time Systems Symposium,* pp. 12-19, Dec. 1989.

[WKSG89] J-Y. Chung W. K. Shih, J. W. S. Liu and D. W. Gillies, "Scheduling tasks with ready times and deadlines to minimize average error," *Technical Reports, UIUCDCS-R-89-1478, Dept. of Computer Science, University of Illinois, Urbana, IL,* 1989.

[ea77] J. K. Lenstra et. al., "Complexity of machine scheduling problems," *Annals of Discrete Mathematics,* 1977

[ea91] S. Baruah et. al., "On-line scheduling in the presence of overload," *in Proceeding of IEEE Foundations of Computer Science Conference,*
pp. 101-110, Otc. 1991.

[ea92] S. Baruah et. al., "On the competitiveness of on-line task real-time task scheduling," T*he Journal of Real-Time Systems,* vol. 4,
pp. 124-144, Jun. 1992.

[vTK91] A. M. van Tilborg and G. M. Koob, eds., *Foundation of Real-Time Computing.* Kluwer Academic Publishers, 1991.