결합허용이 가능한 임베디드 실시간 태스크 관리 메커니즘

정경훈*, 탁성우**, 김창수***

요 약

본 논문에서는 임베디드 실시간 운영체계 수준에서 주기적 및 비주기적 태스크 스케줄링뿐만 아니라 태스크들의 일시적인 결함을 복구할 수 있는 결합허용이 가능한 임베디드 실시간 태스크 관리 메커니즘을 제안한다. 기존의 임베디드 운영체계들은 주기적 및 비주기적 태스크들을 동시에 고려한 스케줄링 메커니즘을 지원하지 않는다. 그리고 태스크 결합 복구 메커니즘의 미지원으로 인해 결합 태스크로 인한 시스템 고장은 야기할 수도 있다. 제안된 결합허용 실시간 태스크 관리 메커니즘은 운영체계 수준에서 주기적 태스크들의 마감시한과 비주기적 태스크의 실형완료를 보장할 뿐만 아니라 일시적인 결함이 발생한 태스크를 복구함으로써 태스크 결합으로 인한 시스템 고장을 방지할 수 있다.

Mechanism for Managing Fault-Tolerant Embedded Real-Time Tasks

Kyunghoon Jung*, Sungwoo Tak**, Changsoo Kim***

ABSTRACT

In this paper, we propose a mechanism for both scheduling the hybrid-task set which consists of periodic and aperiodic tasks and recovering tasks with transient faults on the level of the operating system. Existing embedded operating systems would not provide the scheduling of both periodic and aperiodic tasks. Also because of not supporting the recovery of task failures, they can not prevent system failure from transient task faults. Proposed method, on the level of operating system, is able to not only meet the deadlines of all periodic tasks but also complete the execution of aperiodic tasks. In addition, it is able to prevent the system failure from transient task faults by recovering the task faults.

Key words: Embedded System(임베디드 시스템), Task Scheduling(태스크 스케줄링), Real-Time(실시간), Fault-Tolerance(결합허용)

1. 서 론

산업용 로봇, 공장 자동화와 같은 산업용 제어 시스템이나 자동차 엔진제어, 항공기 운항 제어와 같은 교통제어 시스템을 위한 임베디드 운영체계는 범용운영체계에 달리 제한된 컴퓨팅 자원을 효율적으로 관리하고 외부 반응에 즉각적으로 응답하는 실시간 성을 지원해야 한다. 일반적으로 기존의 상용 및 공개용 임베디드 운영체계들은 실시간 태스크들을 스케줄하기 위해 POSIX 1003.1.B 표준[1]에서 제안한 태스크 스케줄링 방법들을 지원한다. POSIX 1003.1B 는 실시간 요구사항을 만족하는 에뮬레이션을 위
해 Unix 인터페이스를 확장한 API(Application Programming Interface)이며, FIFO(First-In First-Out), 선점형 우선순위 기반 스케줄링과 라운드 로빈 (Round-Robin) 스케줄링을 실시간 테스크 스케줄링을 위한 기본 요구사항으로 제안하고 있다[1]. 그러나 이러한 테스크 스케줄링 기법들은 임베디드 운영체계뿐만 아니라 범용 운영체계에서도 사용되는 기법들이므로 임베디드 실시간 운영체계가 결정성 (deterministic)과 신뢰성(reliability)과 같은 실시간 요구사항을 만족하기 위해서는 실시간 테스크들을 위한 스케줄링 메커니즘을 지원할 필요가 있다. 실시간 테스크는 실행시간의 속성에 따라 실행시간 간격이 일정한 주기적 테스크와 그렇지 않은 비주기적 테스크로 구분할 수 있다. 주기적 테스크들은 마감시간(deadline)내에 실행완료가 보장되어야 하며 그렇지 않을 경우 테스크 오동작으로 인한 시스템 고장(failure)을 유발할 수 있다. 또한 주기적 테스크는 우선순위가 높은 비주기적 테스크로 인해 마감시간을 초과(miss)할 수도 있다. 따라서 주기적 및 비주기적 테스크가 동시에 존재하는 임베디드 시스템에서 테스크들의 실시간성 지원을 위해서 주기적 테스크들의 마감시간과 비주기적 테스크들의 실행완료를 보장하는 실시간 테스크 관리 메커니즘이 필요하다.

의료장비나 자동차 엔진 제어, 항공기 운항 제어와 같이 안전에 민감(safety-critical)한 임베디드 실시간 시스템에서 발생하는 결함(fault)은 인간의 생명에 직접적인 영향을 미치기 때문에 일부 작업의 결함으로 인한 시스템 고장(failure)은 치명적인 결과를 야기할 수 있다. 센서와 마이크로프로세서, 액체레이터(accelerator)로 구성된 자동차의 SBW(Steering-by-Wire) 시스템에서 액체레이터 결함은 자동차 운전자 의지와 상관없이 진행방향을 바꾸어 나가게 할 수도 있다[3]. 따라서 이러한 safety-critical한 임베디드 실시간 시스템을 위한 운영체계는 반드시 작업 결함으로 인한 시스템 고장까지 방지하는 메커니즘을 제공해야 한다. 결함은 결함 지속시간에 따라 영구적 결함(permanent fault), 간헐적 결함(intermittent fault), 일시적 결함(transient fault)으로 분류될 수 있다. 결함에 대한 기존의 결함항용 연구들은 주로 여분(redundancy) 기법들을 사용하고 있으며, 이에는 크게 여분의 하드웨어나 소프트웨어 컴포넌트를 이용하는 공간여분과 결함 발생 시 가까운 기업 중요 성공성 시장부터 작업을 제거하는 rollback하는 시간 여분 기법으로 분류될 수 있다[4].

본 논문은 주기적 및 비주기적 테스크들이 동시에 실행되는 임베디드 실시간 시스템에서 주기적 테스크들의 마감시간과 비주기적 테스크들의 실행완료를 보장함과 동시에 테스크의 일시적인 결함을 복구할 수 있는 결함항용이 가능한 실시간 테스크 관리 메커니즘을 제안하고자 한다.

본 논문의 구성은 다음과 같다. 2장에서는 관련연구를 기술하고, 3장에서는 본 논문에서 제안하는 결합항용 임베디드 실시간 테스크 관리 메커니즘에 대해 기술한다. 4장에서는 구현 결과 및 분석을 기술하며, 마지막 5장에서는 결론을 기술한다.

## 2. 관련 연구


$$U = \sum_{i=1}^{N} \frac{C_i}{T_i} \leq N \left( \frac{1}{N} \right)$$


3. 결합허용 임베디드 실시간 태스크 스케줄러

3.1 가정 및 시스템 구성

본 논문에서 제안하는 태스크 스케줄러를 위해 다음의 가정을 한다.

- 주기적 태스크를 $\tau_i$라고 하고, $\tau_i$의 실행시간과 주기를 각각 $C_i$와 $T_\tau$라고 하면 $n$개의 태스크를 가지는 태스크 집합 $S=\{\tau_1,\tau_2,\ldots,\tau_n\}$에 대해 $\tau_i$는 $\tau_{i+1}$보다 우선순위가 높다.
- 시스템은 하나의 처리기를 가진다.
- 결합의 비용과 스케줄링 오버헤드는 무시할 수 있다.
• 테스크들은 모두 독립적이며, 주기적 테스크는 스스로 수행을 중단하지 않는다.
• 테스크 집합은 RM 알고리즘에 의해 스케줄된다.
• 주기적 테스크의 마감시한과 주기는 동일하다.
• 비주기적 테스크들은 FIFO(Firt-In First-Out) 순서로 스케줄된다.
• 비주기적 테스크의 도착 시간은 알려지지 않으며, 수행시간은 도착 시에 알려진다.
• 주기적 테스크 집합은 초기 시작 시간이 동일한 동기적(synchronous) 테스크 집합이다.


3.2 임베디드 실시간 테스크 스케줄러

가. 스케줄 가능성 검사자

SC는 테스크 집합 내 개별 테스크의 처리기 이용률 계산을 통해 전체 처리기 이용률을 계산한다. 처리기 이용률 계산을 위해 3.1절의 가정을 따르며 본 우어들은 표 1과 같다. 그리고 표 1에 사용된 시간 $t$는 (1)의 $s_i$에 의해 계산된다.

$$s_i = \{ k \cdot T^j = 1, \ldots, k = 1, \ldots, \lfloor T^j / T \rfloor \}$$

(1)
표 1. 처리기 이용률에 사용된 기본 용어

<table>
<thead>
<tr>
<th>용어</th>
<th>정의</th>
<th>의미</th>
</tr>
</thead>
<tbody>
<tr>
<td>( W(t) )</td>
<td>( \sum_{j=1}^{i} C_j \cdot \left[ \frac{t}{T_j} \right] )</td>
<td>테스크 작업량</td>
</tr>
<tr>
<td>( L(t) )</td>
<td>( W(t)/t )</td>
<td>처리기 이용률</td>
</tr>
<tr>
<td>( L_i )</td>
<td>( \min_{0 \leq t \leq \tau_i} L_i(t) )</td>
<td>개별 테스크의 처리기 이용률</td>
</tr>
<tr>
<td>( L )</td>
<td>( \max_{1 \leq i \leq n} L_i )</td>
<td>테스크 전체 집합의 처리기 이용률</td>
</tr>
</tbody>
</table>

개별 테스크의 처리기 이용률은 \((1)\)에서 구한 \( \delta_i \)의 요소 값들에 대해 계산한 \( L_i(t) \)의 값들 중에서 최소값이 \( L_i \)가 된다. \( L \)의 값을 중 최대값인 \( L \)은 테스크 전체 집합의 처리기 이용률로서 \( L \leq 1 \)이면 테스크 집합 내의 모든 처리기의 테스크들이 스케줄 가능하다.
주기적 테스크 집합 \( \tau = (T, C) = \{(5,2), (10,3), (40,4)\} \)가 주어질 때, 표 2는 \( \tau \)에 대한 모든 가능한 처리기 이용률을 나타낸다. 표 2에서 \( L_1 = 0.40, L_2 = \min(1.00, 0.70) = 0.70, L_3 = \min(1.80, 1.10, 1.06, 0.90, 0.92, 0.83, 0.85, 0.80) = 0.80 \)이다. 따라서 \( \tau \)의 전체 처리기 이용률 \( L = \max(0.40, 0.70, 0.80) = 0.80 \)이므로 조건 \( L \leq 1 \)을 만족하여 주기적 테스크 집합 \( \tau \)는 스케줄 가능하다.

나. 여유시간 매니저
여유시간 매니저(STMAT)는 처리기 여유시간 계산을 위해 다음과의 용어를 정의한다. \( \tau \)의 \( j \)-번째 요소를 \( \tau_j \)라고 하면, \( \tau_j \)의 도착시간\( (release \ time) \)은 \((j-1) T_i \)이고, 마감시한은 \( D_j \)로 실행시간은 \( C_j \)가 된다. \( a_i(t) \)은 시간 \([0, t] \), \( 0 \leq t \leq D_j \)에서 우선순위가 \( i \)보다 크거나 같은 비주기 테스크들의 전체 실행시간이기로 하고, \( P_i(t) \)는 \([0, t] \)에서 우선순위 \( i \)인 주기적 테스크의 준비시간이라고 하자. 그리고 \( L_i(t) \)은 \([0, t] \)에서 우선순위 \( i \)보다 높은 테스크들이 수행되거나 처리기가 유류 상태인 시간이라고 하며, 테스크들의 전체 처리기 이용시간 \( W(t) \)는 \((2)\)에 의해 구할 수 있다.

\[
W(t) = a_i(t) + P_i(t) + L_i(t) \tag{2}
\]

\( A_i \)를 시간 \([0, C_j] \)에서 우선순위가 \( i \)보다 크거나 같은 비주기 테스크들을 처리하기 위한 최대 여유시간이라고 하면 \((2)\)는 \((3)\)의 조건식을 얻을 수 있다. 이때, \( C_j \)는 주기적 테스크 \( \tau_j \)의 종료시간이다.

\[
\min_{0 \leq t \leq C_j} \left\{ (A_i(t) + P_i(t))/t \right\} = 1 \tag{3}
\]

\( A_i(t) = A_{i_j}, C_j \leq t < C_j, j \geq 1 \)이라고 하고, \( A^*(t) \)를 최대 여유시간이라고 하며,

\[
A^*(t) = \min_{1 \leq i \leq n} A_i(t), C_i \leq t < C_j \tag{4}
\]
가 된다. 주기적 테스크 집합 \( \tau = (T, C) = \{(5,2), (10,3), (40,4)\} \)가 주어졌고 모든 \( \tau \)의 \( i \leq j \)에 대한 최소 공배수인 hyperperiod를 계산하면 \( 40 \)이 된다. 앞절로부터 \( \tau \)는 스케줄 가능이므로 \((2)\)\~\((4)\)로부터 그림 2의 스케줄 테이블을 얻을 수 있다. 그림 2에서

<table>
<thead>
<tr>
<th>( i )</th>
<th>( \tau_i(T, C_i) )</th>
<th>( t )</th>
<th>( W(t) )</th>
<th>( L(t) )</th>
<th>( L )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>( \tau_1(5, 2) )</td>
<td>5</td>
<td>( C_1 = 2 )</td>
<td>0.4</td>
<td>0.4</td>
</tr>
<tr>
<td>2</td>
<td>( \tau_2(10,3) )</td>
<td>5</td>
<td>( C_2 + C_3 = 5 )</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td>10</td>
<td>( 2C_2 + C_3 = 7 )</td>
<td>0.7</td>
<td>1.0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>( C_1 + C_2 + C_3 = 9 )</td>
<td>1.8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>( 2C_2 + C_3 = 11 )</td>
<td>1.1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>( 3C_2 + 2C_2 + C_3 = 16 )</td>
<td>1.0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>( 4C_2 + 2C_2 + C_3 = 18 )</td>
<td>0.9</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>( 5C_2 + 3C_2 + C_3 = 23 )</td>
<td>0.9</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>( 6C_2 + 3C_2 + C_3 = 25 )</td>
<td>0.8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>35</td>
<td>( 7C_2 + 4C_2 + C_3 = 30 )</td>
<td>0.8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>40</td>
<td>( 8C_2 + 4C_2 + C_3 = 32 )</td>
<td>0.8</td>
<td>0.8</td>
<td>0.8</td>
<td></td>
</tr>
</tbody>
</table>

그림 2. 여유시간 매니져에 의해 생성된 여유시간 테이블(STT)
명암부분은 테스크 실행으로 인해 처리기가 busy인 상태를 의미한다. 표 2로부터 3의 전체 처리기 이용률은 80%이고 hyperperiod는 40이므로 비주기적 테스크 수행을 위해 할당 가능한 처리기 여유시간은 8(20%)임을 알 수 있다. 그림 2에서 비주기적 테스크 AT가 t=18에 도착하였다면 STMAT는 최대 2의 여유시간을 할당할 수 있다.

d. 비주기적 테스크 스케줄러

비주기적 테스크 스케줄러(ATS)는 비주기적 테스크 요구 큐(ARQ: Aperiodic Task Request Queue)에 등록된 비주기적 테스크에 대해 STMAT로부터 STT의 값을 할당받은 비주기적 테스크를 FIFO 방식으로 스케줄한다. 주기적 테스크 집합 $\tau = (T, C) = ((5, 2), (10, 3), (40, 4))$와 실행시간이 9인 비주기적 AT가 주어지며, AT의 우선순위는 $t_2$보다는 작고 $t_3$보다는 크다고 하자. $\tau$로부터 hyperperiod는 40이다. 그림 3은 $\tau$와 AT를 선정해 우선순위 기반의 테스크 스케줄링 시의 테스크 할당 테이블로서 AT는 실행을 완료하지만 $\tau_2$는 AT로 인해 41에 실행을 완료하여 마감시간을 초과한다. 그림 4는 STMAT로부터 여유시간을 할당받아 AT를 스케줄한 경우의 테스크 할당 테이블이다. 그림 4에서 AT는 STMAT로부터 여유시간을 할당받아야만 스케줄되기 때문에 비록 $t_3$보다 우선순위가 높지만 $t_3$의 실행완료 후에 스케줄된다. 왜냐하면 그림 2의 STT에서 할당 가능한 가장 빠른 여유시간은 시각 18이기 때문에 AT는 18에서 최초로 스케줄될 수 있기 때문이다. 그림 4에서 모든 주기적 테스크들은 hyperperiod내에 자신의 실행을 완료하는 반면 AT는 hyperperiod내에 실행완료 할 수 없으나 다음 hyperperiod에서 실행을 완료할 수 있다.

라. 주기적 테스크 스케줄러

주기적 테스크 스케줄러(PTS)는 주기적 테스크와 비주기적 테스크에 대해 선정 가능한 우선순위 기반의 스케줄링을 수행한다. 주기적 테스크들의 우선순위는 RM(6)에 의해 부여되며 비주기적 테스크는 설계단계에서 테스크의 중요도에 따라 우선순위가 정해진다. PTS는 테스크를 스케줄하기 위해 주기적 및 비주기적 테스크 우선순위 테이블을 관리하며, 두 개의 우선순위 테이블에서 가장 높은 우선순위를 가지는 테스크를 선정하여 스케줄한다.

3.3 결합하용 메커니즘

그림 5는 본 논문에서 제안하는 결합하용 메커니즘(FTM: Fault Tolerance Mechanism)으로서, 결합 테스크 관리자(FM: Fault Manager)와 PATS와의 상호 연락성을 가진에서 테스크 결합을 복구한다. 임베드 시스템의 낮은 처리기 성능으로 인해 테스크 결합은 테스크 재시작(task reexecution) 방법을 통해 복구된다. 테스크 실행 중에 결합이 발생하면, FM은 결합 테스크가 비주기적 테스크인지 주기적 테스크인지 판별한다. 만약 비주기적 테스크 결합이면 비주기적 테스크 우선순위 테이블에서 해당 테스크를 삭제한다. 그런 다음 FM은 ATS에게 결합 테스크 재실행을 요청하고, ATS는 STAA로부터 STT의 값을 할당받은 결합 테스크를 스케줄한다. 비주기적

![그림 3. 우선순위 기반에 의한 테스크 할당 테이블](image3)

![그림 4. STMAT를 이용한 테스크 할당 테이블](image4)
그림 5. 결합허용 메커니즘

테스크로 스케줄된 결합 테스크는 PTS에 의해 우선 순위에 따라 재설정되며, 결합 테스크의 복구가 완료된 FM은 복구된 테스크를 비주기적 테스크 우선 순위 태이블에 동록한 후에 복구 작업을 완료한다. 결합 테스크가 주기적 테스크이면, FM은 주기적 테스크 우선 순위 태이블에서 해당 테스크를 삭제하고 스키줄 가능성을 검사를 제로접한다. SC는 결합 테스크를 제외한 주기적 테스크들에 대해 스키줄 가능성을 검사를 수행하고, STCAT는 STT를 생성한다. 그런 다음 비주기적 테스크 결합복구와 동일한 작업을 수행한다. 주기적 테스크 복구 작업의 경우, 복구가 완료되면 FM은 스키줄 가능성을 검사와 STT값을 한 번 더 재산하여 주기적 테스크 결합을 이전의 상태로 복구한다.

4. 구현 및 분석

본 논문에서 제안한 FT-ERTS의 구현을 위해 공개용 엄베디드 실시간 운영체계인 uC/OS ver. 2.76을 소프트웨어 환경으로 사용하며, 인텔 Pentium IV 기반의 Windows XP 환경에서 시뮬레이션을 수행한다. 본 결과는 SC, STMAT, 그리고 실시간 테스크들에 대한 평균 응답시간 계산, 결합 테스크의 복구 과정을 통해 본 논문의 제안 방법을 분석한다. 표 3은

<table>
<thead>
<tr>
<th>우선 순위</th>
<th>테스크 이름</th>
<th>주 기</th>
<th>최악 경우의 수행시간</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>Real_Time_Clock</td>
<td>50</td>
<td>0.28</td>
</tr>
<tr>
<td>4</td>
<td>Read_Bus_IP</td>
<td>10</td>
<td>1.76</td>
</tr>
<tr>
<td>9</td>
<td>Command_Actuators</td>
<td>200</td>
<td>2.13</td>
</tr>
<tr>
<td>10</td>
<td>Request_DSS_Data</td>
<td>200</td>
<td>1.43</td>
</tr>
<tr>
<td>11</td>
<td>Request_Wheel_Speeds</td>
<td>200</td>
<td>1.43</td>
</tr>
<tr>
<td>7</td>
<td>Request_IRES_data</td>
<td>100</td>
<td>1.43</td>
</tr>
<tr>
<td>6</td>
<td>Process_IRES_data</td>
<td>100</td>
<td>8.21</td>
</tr>
<tr>
<td>8</td>
<td>Control_Law</td>
<td>200</td>
<td>52.84</td>
</tr>
<tr>
<td>13</td>
<td>Process_DSS_data</td>
<td>1000</td>
<td>5.16</td>
</tr>
<tr>
<td>12</td>
<td>Calibrate_Gyro</td>
<td>1000</td>
<td>6.91</td>
</tr>
</tbody>
</table>

표 4. STMAT 예제 테스크 집합(시간단위: ms)

<table>
<thead>
<tr>
<th>테스크 우선순위</th>
<th>주 기</th>
<th>최악 경우의 수행시간</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>10</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>40</td>
<td>4</td>
</tr>
<tr>
<td>6</td>
<td>50</td>
<td>5</td>
</tr>
<tr>
<td>7</td>
<td>50</td>
<td>6</td>
</tr>
<tr>
<td>8</td>
<td>100</td>
<td>13</td>
</tr>
<tr>
<td>9</td>
<td>100</td>
<td>9</td>
</tr>
<tr>
<td>10</td>
<td>200</td>
<td>15</td>
</tr>
<tr>
<td>11</td>
<td>200</td>
<td>5</td>
</tr>
<tr>
<td>12</td>
<td>200</td>
<td>10</td>
</tr>
<tr>
<td>13</td>
<td>200</td>
<td>9</td>
</tr>
</tbody>
</table>
Olympus 위성에서 실행된 고도 및 궤도 제어 시스템 (AACS: Attitude and Orbital Control System) [17]의 테스크들로서 SC를 검증하기 위한 테스크 집합이다. 표 4는 STTT의 수행을 검증하기 위한 테스크 집합으로서, 10개의 주기적 테스크들로 이루어져 있으며 테스크들의 우선순위는 RM에 의해 할당된다.

그림 6은 표 3의 AACS의 테스크들에 대한 스케줄 가능성 검사를 수행한 결과로서 표 3의 테스크 집합에 대한 최대 처리기 이용률 $L$은 0.5792(<1)이므로 스케줄 가능성을 알 수 있다. 그림 7은 표 4의 테스크 집합에 대한 STTT로서 표 4의 테스크 집합에 대한 hyper period는 200이므로 STTT의 slack time의 크기는 200이다. STTT의 값이 0보다 크면 그 시간에 서의 비주기적 테스크 실행을 위한 할당 가능 처리기 여유시간이 존재함을 의미한다. 그림 7에서 $t=185$일 때 비주기적 테스크 요청이 존재하면 slack_time[185]의 값이 13이므로 최대 13의 처리기 여유시간을 비주기적 테스크에게 할당가능하다.

그림 8은 ATS에 의해 생성된 비주기적 테스크의 실행을 나타낸다. 주기적 테스크 집합은 표 4의 테스크들로 사용하였으며, 비주기적 테스크의 실행시간은 13으로 하고 우선순위는 21로 한다. $t=185$에서 비주기적 테스크 요청이 도착하면 그림 7로부터 slack_time[185] = 13임을 알 수 있으므로 그림 8(a)와 같이 13의 처리기 여유시간을 비주기적 테스크 실행시간으로 할당 받는다. 생성된 비주기적 테스크가 수행을 완료하면 그림 8(b)와 같이 종료 메시지 “Aperiodic Task 21 completed”을 출력하여 테스크 수행을 완료함을 알 수 있다.

그림 9는 주기적 테스크(Periodic Task 6)에서 결합이 발생할 경우의 결합 복구가 Periodic Task 6에서 결합이 발생하기 그림 9(a)와 같이 Periodic...
Task 6은 우선순위 테이블에서 삭제된다. 그림 다음, 그림 9(b)와 같이 결합 허용 메커니즘을 통해 테스크 결합을 복구한다. 그림 9(b)의 경우, 결합 테스크의 복구 작업은 $t=185$에서 실행되었으며, 이때 할당받은 처리기 여유시간은 13임을 알 수 있다. 복구된 Periodic Task 6은 주기적 테스크 우선순위 테이블에 재등록된 후 정상적으로 작업을 수행할 수 있다.

그림 10은 처리가 이용률이 80%인 주기적 테스크 집합에 대해 비주기적 테스크가 도착하였을 경우, 본 논문에서 제안한 방법을 사용한 경우와 그렇지 않은 경우에 대해 수행한 비교의 결과이다. 비주기적 테스크의 응답시간 비교를 위해 비주기적 테스크 집합의 처리가 이용률을 각각 5%, 10%, 15%, 20%로 하여 그에 대한 비주기적 테스크의 평균 응답시간을 측정하였다. 그림 10으로부터 본 논문에서 제안한 심사 시간 테스크 관리 메커니즘의 기존의 주기적 및 비주기적 실시간 속성을 고려하지 않은 방법보다 빠른 응답시간을 가짐을 알 수 있다. 따라서 빠른 응답시간을 필요로 하는 비주기적 테스크가 존재하는 일바디드 실시간 시스템에서 본 논문의 제안 방법을 응용할 경우 비주기적 테스크 처리 시간을 단축시킬 수 있을 것이다.

5. 결론

본 논문은 일바디드 실시간 운영체계에서 주기적 및 비주기적 테스크의 실시간성을 보장함과 동시에 일시적인 테스크의 결합을 복구할 수 있는 결합허용 일바디드 실시간 테스크 관리 메커니즘을 제안하고 구현하였다. 기존의 일바디드 실시간 운영체제들은 운영체제 수준에서 실시간성을 지원하지 않거나, 실시간 스케줄링을 지원하더라도 주기적 테스크 또는 비주기적 테스크를 위한 API만 지원하기 때문에 주기적 테스크들의 스케줄 가능성이 퇴색하게 된다. 그리고 기존의 일바디드 운영체계들은 의료장비나 항공운항 체어와 같은 안전에 민감한 분야에서 발생하는 테스크 결합을 복구할 수 있는 메커니즘은 연구도 많지 않다. 따라서 제안된 테스크 관리 메커니즘은 운영체제 수준에서 주기적 테스크의 마감시간과 비주기적 테스크들의 응답성을 보장하고 실시간 테스크의 일시적인 결합을 복구할 수 있는 기능을 제안하고 있다. 이를 통해 일바디드 실시간 시스템의 신뢰성을 향상
시키고 테스크 결합으로 인한 시스템 고장을 예방할 수 있는 방법을 제시하고 있다.

참고문헌


정 경 준
1996년 부경대학교 전자계산학과 (학사)
1998년 부경대학교 전자계산학과 (석사)
2006년 부경대학교 전자계산학과 (박사)
2006년~현재 부산대학교 U-Port 정보기술산학동사년간 전임연구원
관심분야: 셀서버트워크, 엄베디드 운영체제, 실시간 스케줄링, 분산병렬처리

김 창 수
1984년 울산대학교 전자계산학과 (학사)
1986년 중앙대학교 컴퓨터공학과 (석사)
1991년 중앙대학교 컴퓨터공학과 (박사)
2002년~2003년 UMKC 방문교수
1962년~현재 부경대학교 전자컴퓨터정보통신공학부 교수
관심분야: USN 방제시스템, LBS/GIS, Semantic GIS 검색, 엄베디드 시스템

탁 성 우
1995년 부산대학교 컴퓨터공학과 (학사)
1997년 부산대학교 컴퓨터공학과 (석사)
2003년 미국미주리주립대학교
Computer Science (박사)
2004년~현재 부산대학교 정보컴퓨터공학부 교수
2004년~현재 부산대학교 컴퓨터정보통신연구소 겸임 연구원
관심분야: 유무선 네트워크, SoC 설계, 실시간 시스템, 위치인식, 최적화 기법, 그래프 이론, 큐잉 이론