Safety-Critical Real-Time Operating System의 설계 및 구현*

윤기현°, 김용희, 박희상, 성영락†, 이철훈
충남대학교 컴퓨터공학과
(khyoon°, yonghee, hspark, chlee)@ce.cnu.ac.kr
† 국민대학교 전자정보통신공학부
yeong@mail.kookmin.ac.kr

Design and Implementation of Safety-Critical Real-Time Operating System

Ki-Hyun Yoon°, Yong-Hee Kim, Hee-Sang Park, Yeong Rak Seong†, and Cheol-Hoon Lee
Dept. of Computer Engineering, Chungnam National Univ.
† School of Electrical Engineering, Kookmin Univ.

요 약

실시간 운영체제(Real-Time Operating System)는 그 실행환경상 시스템이 예상치 못한 특정 이벤트가 발생하는 악 조건속에서도 태스크 수행의 데드라인을 초과하지 않도록 시간적인 측면의 determinism 을 보장하는 안정된 스케쥴링 기능을 갖춘 운영체제이다. 또한, 예상치 못한 fault 를 미연에 방지할 수 있는 장치를 가지고 있어야 한다. 본 논문에서는 예상치 못한 fault 를 미연에 방지하기 위하여 필요한 운영체제를 설계 및 구현하는데 그 목적이 있다.

1. 서 론

실시간 운영체제(Real-Time Operating System, 이하 RTOS)는 실행환경상 시스템이 예상치 못한 특정 이벤트가 발생하는 악조건 속에서도 태스크 수행의 데드라인을 초과하지 않도록 시간적인 측면의 determinism을 보장하는 안정된 스케쥴링 기능을 갖춘 운영체제이다. 또한, System의 수행중에 발생한 예상치 못한 fault에 대해서도 처리할 수 있도록 구현되어야 한다. 일반적으로 RTOS를 사용하는 Embedded System의 경우에는 예상치 못한 fault에 대한 처리가 미흡 하여도 System에 큰 영향을 미치지 않는 경우가 대부분 이지만 비행기의 자동항법 시스템, 군사용 제어를 목적으로하는 Embedded System의 경우 Critical System에서는 예상치 못한 fault를 처리하지 못할 경우 심각한 문제가 발생할 수 있기 때문에 이를 처리 할 수 있는 장치가 구비 되어야 한다. Fault에 대한 처리를 application level에서 제어 하는 것이 대부분이지만 RTOS상에서도 예상치 못한 fault가 발생하지 않도록 하는 루틴이 필요하게 된다. 이와 같이 fault가 발생하지 않도록 구현된 RTOS를 Safety-Critical RTOS라고 한다.[1][2]

본 논문에서는 Safety-Critical RTOS를 어떻게 설계, 구현하였는지를 기술하고 있다. 2 장에서는 관련연구로 Safety의 의미에 대해 기술하고, 3 장에서는 Safety-Critical RTOS가 갖추어야 할 요건에 대해 설명하였고, 4 장에서는 실제로 Safety-Critical RTOS가 어떻게 구현되어있는지에 대해 기술하였으며, 마지막으로 5 장에서는 결론 및 향후 연구과제에 대하여 논하였다.

2. 관련연구

2.1. Reliability vs. Safety

Reliability는 주어진 시간 내에서 얼마나 시스템이 주어진 기능을 얼마나 에러 없이 수행하는가를 확률적으로 나타내는 것이다. 이에 반해 Safety는 에러의 발생 여부에 상관 없이 시스템이 주어진 기능을 수행해 내는 것을 의미한다. 어떤 System이 안전하다고 할때는 다음 두가지 조건을 반드시 만족해야 한다.

- ✓ Fault가 존재하지 않을 때, System내에 어떤 위험성도 존재하지 않는다.
- ✓ System내의 single component에서 fault가 발생하였을 때, 시스템이 안정적으로 동작해야 한다.

3. Safety-Critical RTOS의 요건

3.1. Memory Protection

프로세스기반의 운영체제들은 프로세스마다 서로 다른 메모리 영역을 사용하므로 프로세스간의 메모리 영

* 본 논문은 산업자원부 중기거점과제 연구비에 의해 지원되었음

역을 침범하지 않도록 MMU를 이용하여 메모리간의 영역을 보호한다. 그러나, 일반적인 RTOS에서 사용하고 있는 태스크 기반의 운영체제에서는 메모리를 공동으로 사용하므로, 각 태스크의 메모리를 보호하기가 어렵다. 따라서 별도의 메모리 보호방법을 사용하지 않는다. 다만, 메모리 상의 전역 변수(구조체)를 보호하기 위해서 MAGIC NUMBER를 사용한다. Magic number는 각 구조체마다 독특한 값을 가지게 되어, 해당 메모리가 더럽혀졌을 경우 에러 체크를 하기 위해 사용한다.

태스크 기반의 RTOS에서 꼭 필요한 메모리 보호 방법은 각 태스크의 스택 영역을 보호 하는 것이다. 이는 모든 태스크들이 분리된 메모리 영역을 사용하지 않고, Stack에는 각 태스크의 Context가 들어있기 때문에 stack overflow가 발생하게 되면 시스템에 치명적인 영향을 끼치기 때문에 이에 대한 처리가 필요하다.

3.2. 부적당한 System Call에 대한 처리

System Call을 사용할 때 부적당한 Parameter를 사용하여 시스템 오류를 발생할 수 있기 때문에 System Call내에 부적당한 Parameter에 대한 처리를 해 주어야 한다.

3.3. Interrupt latency의 최소화

RTOS에서 Critical Region을 보장하기 위해서 주로 사용하는 방법은 Interrupt를 Disable하는 방법이다. 그러나, Critical Region이 크게 되면 Interrupt Latency가 커져 시간적인 *determinism*을 저해할 우려가 있기 때문에 신중하게 사용되어야 한다. 따라서, Interrupt Latency를 최소화 하기 위해서 Interrupt Disable을 사용하는 부분을 최소화 해야 한다.

3.4. Priority Inversion의 방지

*Priority Inversion*이란 하위 레벨의 우선순위를 가지는 태스크가 resource를 가지고 있는 상태에서 상위 레벨 우선순위 태스크에 의해서 *preemption* 되었을 때 최상위 레벨의 태스크가 해당 자원을 요구하게 되는 경우 최상위 태스크는 자원을 가지고 수행이 이루어져야 하지만 상위 레벨 태스크가 수행을 끝 마칠 때까지 하위 레벨 태스크가 수행 되지 못하므로, 상위 레벨 태스크가 수행을 끝마치고, 하위 레벨 태스크가 resource를 해제할 때 까지, 최상위 레벨 태스크가 수행하지 못하는 상황을 의미한다. 이를 해결하기 위해서 Priority Inheritance 혹은 priority ceiling방식을 사용하여 *Priority Inversion*을 방지한다.

3.5. Fault Tolerance

Fault tolerance 시스템은 시스템이 수행중에 Fault가 발생하였을 경우 이를 복구하고 수행 할 수 있는 시스템을 의미한다. 이때, 일반적으로 중복을 두어서 한 부분이 오류를 발생하였을 때 다른 부분이 이를 대신하는 것을 의미한다.

4. Safety-Critical RTOS의 설계 및 구현

4.1. Memory protection의 구현

메모리를 보호하기 위해서 앞에서 제시한 두가지 방법중 전역변수 특히 구조체의 값을 보호하기위해 magic number를 사용하였다. 구조체의 맨 앞에 위치시켜 값이 변형 되었을 경우 잘못된 사용을 방지하기 위해 사용하였다.

```
#define MK_TASK_MAGIC 0xF3CD03E2L

typedef struct mk_task_struct {
    UINT      t_Magic;      /* MAGIC NUMBER */
    ..........................
    ..........................
}MK_TASK;
```

표 1 Magic Number 를 이용한 구조체의 보호

또한 stack overflow를 방지 하는 것이 태스크 기반 RTOS의 memory protection에서 중요하기 때문에 다음과 같이 구현 하였다.

```
MK_Check_Stack
    LDR     a1,[pc,#Current_Thread-.-8]
    LDR     a1,[a1,#0]
    ..........
    LDR     a3,[a1,#&24]
    CMP     sp,a3
    BLT     TCT_Stack_Range_Error
    LDR     a2,[a1,#&28]
    CMP     sp,a2
    BLE     TCT_Stack_Range_Okay
    ..........

MK_Stack_Range_Error
    STR     lr,[sp, #4]!
    MOV     a1,#3
    LDR     a4,=ERC_System_Error
    BX      a4

MK_Stack_Range_Okay
    SUB     a4,sp,a3
    CMP     a4,#80
    BCS     TCT_No_Stack_Error
    STR     lr,[sp, #4]!
    MOV     a1,#3
    LDR     a4,=ERC_System_Error
    BX      a4

..........
```

표 2 Stack Overflow 방지 algorithm 의 구현

4.2. 부적당한 System Call에 대한 처리

부적당한 System Call에 대한 처리위해서는 모든 System Call의 내부에서 Parameter에 대한 예외 처리를 구현 하였다.

4.3. Interrupt Latency의 최소화 방법 구현

Interrupt Latency를 최소화 하기 위해서 Interrupt Disable 할 수 있는 부분을 전역변수가 사용되는 영역으로 한정 하였으며, Signal처리 및 Interrupt Service Routine에서 사용될 가능성이 없는 Critical Region에 대해서 Interrupt Disable을 사용하지 않고 Critical Region을 보장할 수 있는 Schedule LOCK/UNLOCK함수를 구현 하였 사용하였다.

```
Int MK_ContextSwitchDisable(void)
{
    int OldFlags = MK_ContextSwitchFlags;
    MK_ContextSwitchFlags = FALSE;

    return OldFlags;
}

void MK_ContextSwitchRestore(int ContextSwFlags)
{
    MK_ContextSwitchFlags = ContextSwFlags;
}
```

표 3 Schedule Lock/Unlock 함수의 구현

이때 Scheduling Routine에서 Scheduling이 Lock 되어 있을 경우 scheduling을 하지 않는 부분이 포함된다.

4.4. Priority Inversion 문제 해결

Priority Inversion의 경우 시간적은 determinism을 저해할 우려가 있기 때문에 Priority Inversion문제가 발생하지 않도록 Kernel Level에서의 구현이 필요하다. Priority Inversion문제를 해결하기 위하여 많이 사용하는 방법으로는 Priority Inheritance방식과 Priority Ceiling방식이 있다. Priority Inheritance방식은 현재 의 Priority에서 동작하다가 높은 priority의 태스크가 자원을 요구할 경우 리소스를 가지고 있는 태스크의 priority를 요청한 태스크의 priority 까지 올려주는 방식이다. 그러나, 이 경우에는 Priority를 여러 번 높여주는 단점이 있기 때문에 Priority Ceiling 방식을 많이 사용한다.

Priority Ceiling의 경우에는 자원을 할당 받을 때 정해진 priority 까지 올려줘서 자원의 사용을 빨리 끝낼 수 있도록 해주는 방식으로서 본 논문에서는 이 방식을 사용하였다.[4][5]

```
int MK_SemaphorePend(MK_SEM *Sem, mk_time_t Ticks)
{
  ..........
  MK_ChangePriority(MK_CurrentTask, Sem->ceilPrio);

  MK_Restore(ps);
  return(Status);
}
```

표 4 Priority Ceiling 방식의 구현

위의 코드에서 보는 바와 같이 자원을 얻은 후에 해당 태스크의 Priority까지 올려주어 빨리 처리하고 자원을 반납할 수 있도록 해준다. 세마포를 반납하는 함수인 MK_SemaphorePost에서는 올려줬던 Priority를 이전 레벨로 돌려주는 코드가 존재한다.

5. 결론 및 향후 연구과제

본 논문에서는 Safety-Critical Operating System을 구성하기 위해 Memory protection, 부적절한 System Call에 대한 처리, Interrupt Latency의 최소화, Priority Ceiling에 대하여 설계 및 구현 하였다. 그러나, 이외에 Safety-Critical의 요건으로는 태스크들이 주어진 조건에서 데드라인을 어기지 않을 것인가의 여부, 요청된 resource를 할당 받아서 사용이 가능한가(starvation되지 않는지)의 여부등에 대해서도 보장 되어야 하는데 이 부분에 대한 추가적인 연구가 필요하다.

6. 참고문헌

[1] Bruce Powel Douglas, " Safety-Critical Embedded Systems" , Embedded System Programming, Oct. 1999.

[2] David Kleidermacher, Mark Griglock, *Safety-Critical Operating Systems*, Embedded System Programmings, Sep. 2001,

[3] Jean J. Labrosse, *uC/OS : The Real-Time Kernel*, R&D Publications, 1992.

[4] Orv Balcom, *Simple Task Scheduler Prevents Priority Inversion*, Embedded System Programming, Jun. 1995.

[5] 안희중, 박희상, 이철훈," Design and Implementation of Mutual Exclusion Semaphores Using the Priority Ceiling Protocol" , 정보처리학회 추계학술발표 논문집, Vol. 9, No. 2, pp. 555-558, Nov. 2002.