ARX에서의 Pthread Library 구현

김세화
서울대학교 전기공학부 실시간 운영체제 연구실

Pthreads implementation on ARX

Saehwa Kim, RTOS Lab, SoEE, SNU

요 약

본 논문은 실시간 운영체제인 ARX의 독특한 메커니즘을 바탕으로 한 POSIX의 thread library(Pthread library)의 구현에 대하여 다룬다. ARX Pthread library는 POSIX 1003.1c의 규약을 따르며 실시간 프로그래밍의 특성에 맞도록 이를 확장한 주기적인 thread, deadline handler 등의 확장된 인터페이스를 제공한다. 이와 더불어 정확한 실시간 특성을 제공하기 위하여 사용자 수준 타이머의 관리도 제공하고 있다. 한편 ARX의 핵심인 가상 쓰레드와 개선된 스케줄링 이벤트 업콜 방식을 바탕으로 사용자 수준 쓰레드와 사용자 수준 스케줄러의 동기화를 제공하는 사용자 수준 동기화 도구를 구현하였다. 이러한 Pthread library를 사용한 응용프로그램의 timing anlysis의 결과는 ARX의 뛰어난 실시간 처리 능력과 함께 다양한 실시간 응용 프로그램의 요구를 수용할 수 있음을 보여준다.

1. 서 론

쓰레드는 동시성을 표현할 적절한 수준의 추상화 방법을 제공할 수 있고, 효과적인 자원의 관리와 빠른 문맥교환 등의 장점 등을 가지고 있다. 그 결과 응용 프로그램을 조직하고 동시성을 표현하는 필수적 메커니즘으로 발전하였다. 이에 따라 POSIX에서도 1003.1c에서 thread에 관한 표준 규약을 정의하였으며, 이를 통상 Pthread라고 한다.

ARX는 실시간 운영체제로서, 커널에서 가상 쓰레드와 개선된 스케줄링 이벤트 업콜 방식을 제공함으로써 내장 실시간 시스템에 적합한 새로운 사용자 수준 쓰레드를 지원한다.[2]

본 논문에서는 이러한 ARX의 메커니즘을 바탕으로 한 Pthread Library의 구현에 대하여 다룬다

2. 본 론

ARX Pthread library는 POSIX 1003.1c[1]의 표준을 따르며, 이와 더불어 실시간 환경에 적합한 확장된 쓰레드 인터페이스를 제공한다. 또한 정확한 실시간 특성을 제공하기 위하여 사용자 수준 타이머의 관리도 제공하고 있다. 사용자 수준 쓰레드와 사용자 수준 스케줄러의 동기화를 제공하는 사용자 수준 동기화 도구에 대하여서도 설명한다.

2.1 주기적 실시간 쓰레드를 위한 쓰레드 인터페이스 확장

```
periodic_thread()
{
  thread_set_releasetime(0);
  thead_set_period(10);
  thread_set_deadline(9);
  thread_set_deadline_handler(
                        handler);
  thread_barrier_wait(b,1);
  while(1) {
    /* do something */
      thread_wait_next_period(
                        NULL);
  }
}
```

(A)

```
periodic_thread()
{
  while(1) {
    clock_gettime(&t1);
    /* do something */
     ...
    clock_gettime(&t2);
    if (t2 - t1) > 9)
    /* handle deadline miss */
    delay(10 - (t2 - t1));
  }
}
```

(B)

그림 1 쓰레드 패키지의 주기적 쓰레드 지원을 가지고 작성한 주기적 쓰레드 (A)와 전통적 시스템에서 이러한 지원 없이 작성된 코드 (B)

POSIX 1003.1c[1]은 선점 우선순위 스케줄링, 타이머 관리와 우선순위 계승과 같은 실시간 시스템에 관련된 많은 특징들을 포함한다. 그러나, 실시간 프로그램에서 요구하는 주기, 데드라인, 펠리스 타임과 초기 위상 등의 시간 속성을 명시하는데 필요한 특징들은 결여되어 있다. 또한, 주기적 실시간 쓰레드의 정확한 행동 방식을 정의하지 않고 있다. 이들 없이, Pthread 라이브러리로 구현된 주기적 쓰레드들은 심각한 지터(jitter)의 영향을 받기 쉽다. 게다가 Pthread는 주기적 쓰레드를 지원하지 않으므로, 프로그래머는 그림 1(B)와 같이 임시 변통 적인 방법으로 코드를 작성해야 한다. 이러한 문제를 해결하기 위하여 시간 속성을 제공하는 인터페이스를 가지고 Pthread 라

이브러리를 확장 구현했다. 이들 함수의 원형이 표 1과 같다.

```
int   thread_set_releasetime(release);
int   thread_set_period(period);
int   thread_set_deadline(deadline);
int   thread_set_deadline_handler(handler);
int   thread_wait_next_period();
int   thread_wait_next_period(time_info);

int   thread_barrier_init(barrier);
int   thread_barrier_wait(barrier, initial phase);
int   thread_barrier_release(barrier);
```

표 1. 주기적 쓰레드를 관리하기 위한 API들

주기적 쓰레드 τ_i의 시간 행위는 네 개의 파라미터 (Φ_i, r_i, d_i, p_i)로 기술된다. 여기서 Φ_i는 초기 위상, r_i는 릴리스 타임, d_i는 데드라인이고 p_i는 주기이다(그림 2). 마지막 세 개의 파라미터는 각각 thread_set_releasetime(), thread_set_deadline()과 thread_set_period()로 지정된다. 주기적 쓰레드의 초기 위상은 *barrier* 라는 개념적인 동기화 객체로 명시된다. thread_barrier_wait()을 호출한 주기적 쓰레드들은 다른 쓰레드 -보통 main 쓰레드-가 thread_barrier_release()를 부를 때까지 하나의 장벽(barrier)에서 기다린다. 이 방식에 의해, 임시로 관계를 맺은 모든 주기적 쓰레드들은 동기화된 수행을 시작할 수 있다. 그림 1(A)는 시간 속성 (1, 0, 10, 9)을 가진 주기적 쓰레드의 전형적인 코드를 보여준다. 초기 위상이 1이므로, 첫 번째 주기 인스턴스는 초기 위상에서 1 클락 틱 이후에 시작한다.

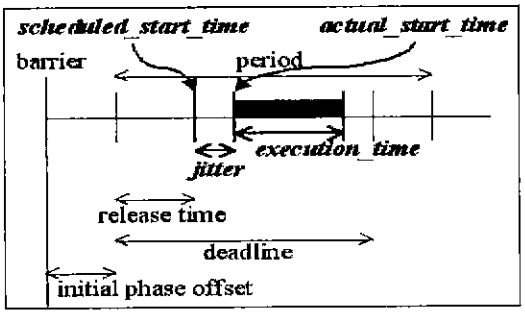

그림 2 쓰레드의 스케줄링 파라미터들

주기적 실시간 쓰레드의 시간 속성이 지정되면, 사용자 수준 스케줄러는 사용자 수준 타이머 방식을 이용하여 매 주기마다 쓰레드를 실행한다. 사용자 수준 타이머는 다음절에서 자세히 설명한다. 이러한 주기적 실행의 일부분을 thread_wait_next_period()가 담당한다 이 함수는 다음 주기의 쓰레드 도착 시간을 계산할 수 있도록 사용자 수준 스케줄러에게 현재 인스턴스가 완료했음을 알려준다. thread_wait_next_period()는 다음 주기까지 수행을 멈추며 만약 이 함수를 호출하기 전에 쓴 시간이 한 주기 이상을 넘어가면 그 만큼 해당하는 다음 주기에 ready 상태가 된다. 이때 놓친 주기의 횟수를 리턴 값으로 넘긴다. 이전 주기의 수행 정보에 대한 값을 period_info structure 타입의 time_info를 통해서 받을 수 있다.

period_info structure는 다음과 같은 3가지 멤버를 가지고 있다(그림 2).

- prev_scheduled_start_time 이전 주기에서 스케줄되어 수행을 시작하기를 희망하는 시스템 timer 시각으로서 tick단위의 값이다.
- jitter 이전 주기에서 실제 스케줄러에게 스케줄되어 수행을 시작한 시각과 스케줄 희망 시각과의 차이 시간이다.
- prev_execution_time 이전 주기에서 쓰레드가 스케줄되어 수행을 시작하여 cpu를 thread_wait_next_period()에 의하여 cpu를 내어놓기 전까지의 수행 시간이다.

시스템에 순간적인 과부하가 지워지면, 주기적 쓰레드는 데드라인까지 수행을 끝마치지 못하는 경우가 발생한다. 이러한 순간적인 시간 결점을 처리하기 위해, 프로그래머는 thread_set_deadline_handler (handler)를 통해 데드라인 미스 핸들러를 등록할 수 있다. 예를 들면 데드라인 미스 핸들러에서 사용자는 데드라인이나 주기 등을 재설정할 수 있다

2.2 사용자 수준 타이머의 관리

내장 실시간 시스템에서 높은 해상도의 타이머 서비스의 제공은 매우 중요하다. ARX 운영체제는 100 μs의 기본 시간 간격을 가진 프로그램 가능한 타이머를 제공한다. 프로그래머는 clock_setres() 시스템 콜로 타이머 시간 간격을 바꿀 수 있다.

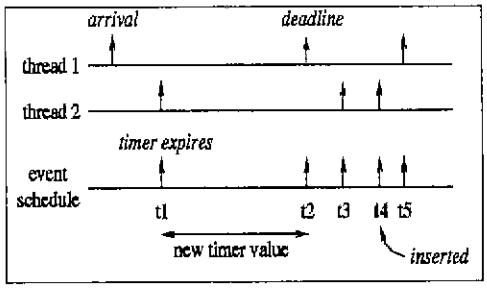

그림 3 타임아웃 이벤트 리스트의 관리

또한 각 프로세스는 사용자 수준 스케줄러가 관리하는 자신만의 사용자 수준 타이머(SCB의 timer 필드)를 갖는다. 사용자 수준 스케줄러는 timer 필드를 다음 타임아웃 값으로 세트한다. 타이머 만기시, 커널은 timer 필드를 지우고, TIMER 엄콜[2]을 만들어 사용자 수준 스케줄러를 실행한다. 사용자 수준 스케줄러는 timer를 다음 타임아웃 값으로 세트한다. 이를 목적으로, 사용자 수준 스케줄러는 쓰레드 도착 시간, 데드라인과 깨어날 시간을 포함하고 있는 타이아웃 이벤트의 리스트를 관리한다. 두 개의 주기적 태스크 τ_1와 τ_2에 대해, 그림 3은 사용자 수준 스케줄러가 어떻게 리스트를 관리하는지를 보여준다. 시간 t_1의 TIMER 엄콜 이전에, 리스트는 t_2와 t_5를 갖고 있다. 시간 t_1에 τ_2가 도착한 후, 사용자수준 스케줄러는 t_2를 다음 타임아웃 이벤트로 기록하고 t_3(새로 도착한 τ_2의 데드라인)와 t_4 (τ_2의 다음 도착시간)를 리스트에 넣는다.

시스템 클락을 알기 위한 비싼 시스템 콜을 피하기 위해서, 커널은 시스템 시간을 SCB의 current system time 필드에 기록한다. 사용자 수준 쓰레드는 단순히 필드의 값을 읽어서 현재 시스템 시간을 얻을 수 있다. 커널이 매 클락 틱마다 필드를 갱신하지만, 단지 현재 실행중인 프로세스의 필드만을 갱신하기 때문에 과부하가 거의 없다.

2.3 사용자 수준 동기화 도구

사용자 수준 쓰레드와 사용자 수준 스케줄러의 실행을 동기화하기 위해서 아래의 설명과 같이 lock 변수를 사용한다.

- 쓰레드는 런 큐와 같은 공유 데이터 구조를 사용할 때 임계 영역에 들어가게 되면 lock 변수를 증가시킨다. 쓰레드는 임계 영역을 빠

```
lock_scheduler()
{
    (*usched_lock)++;
}

unlock_scheduler()
{
    if(--(*usched_lock))
        return;
    if(*usched_lock == 0 & *head != *tail){
        *usched_lock = 1;
        if(setjmp(curthread) == 0){
            stack pointer = u.uscheduler's one
            *sched_lock = 0;
            thread_scheduler();
        }
    }
}
```

그림 4 동기화 primitives

저나갈 때 lock 변수를 감소시키고, 그 값이 0이 되고 이벤트 큐가 비어있지 않으면 사용자 수준 스케줄러를 실행시킨다.

- 만약에 lock 변수가 0이 아닐 경우에 사용자 수준 스케줄러가 불린다면, 스케줄러는 이벤트에 의해서 인터럽트된 쓰레드를 계속 수행시킨다. 왜냐하면 스케줄러는 변수 값이 0이 되기 전까지는 실행을 계속할 수 없기 때문이다.

lock 변수를 사용자 수준에 둠으로서 위의 구현이 가능하다. 그러나 사용자 수준 스케줄러와 사용자 수준 쓰레드 사이의 문맥 교환의 부하를 줄이기 위해서 스케줄러 컨트롤 블럭에 lock 변수를 두었고, 커널이 사용자 수준 스케줄러를 수행시키는 것이 아니라 스케줄러의 lock flag가 0이 아닐 때 인터럽트와 쓰레드를 계속 수행시키도록 하였다.

이제 다른 사용자 수준 쓰레드로부터의 쓰레드 패키지를 보호하기 위한 동기화 도구의 구현에 대해 자세히 설명하겠다. lock과 unlock 도구는 다음과 같다.

- **Lock primitive**

임계 영역에 들어가기 전에 쓰레드는 lock 변수를 증가시킨다.

- **Unlock primitive**

함수가 임계 영역에서의 수행을 끝냈을 때 lock 변수를 감소시킨다. 만약에 값이 0이고 지연된 업콜이 있으면 쓰레드는 재스케쥴되는 것을 위해 문맥을 저장하고, lock 변수를 리셋한 후, 사용자 수준 스케줄러를 수행시킨다. 그렇지 않으면 그냥 리턴한다.

이런 도구들의 구현이 그림 4에 나타나 있다. unlock 도구들은 큐에 지연된 이벤트들이 있거나, 업콜 때문에 커널에 의해서 선점되었을 때 사용자 스케줄러를 부르는 것을 빼먹지 않도록 설계되어야 한다. 그림 4의 unlock_scheduler()는 이런 특징을 갖는다. non-emptiness를 검사하는 중 이벤트가 발생하면(결과적으로 마지막 인덱스가 하나 증가하면), 사용자 수준 스케줄러는 그것이 첫 번째 이벤트(예. empty queue)이면 커널에 의해서 불리고, 큐가 비어있지 않으면 이 함수에 의해서 불리게 된다. 두 번째 경우 인덱스가 커널에 의해서 바뀌기 전에 예전의 마지막 인덱스 값이 로드되었더라도 non-emptiness를 검사는 것은 영향을 받지 않는다.

이 구현에 따르는 영향은 큐에 지연된 이벤트가 없더라도 사용자

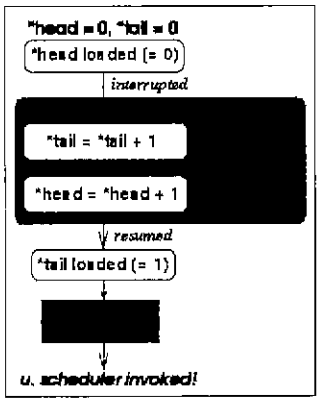

그림 5 불필요한 사용자 수준 스케줄러의 호출 시나리오

수준 스케줄러가 쓰레드에 의해서 불릴 수 있다는 것이다. 이것은 non-emptiness의 검사가 atomic하게 수행될 수 없기 때문이다. 예를 들면 그림 5에서 보듯이 head 값이 레지스터에 저장되었지만 tail 인덱스는 TIMER 이벤트 때문에 아직 저장되지 못한 경우이다. 시작할 때 head와 tail의 값은 모두 0이다. 그러나 TIMER 이벤트가 처리되는 동안 커널과 사용자수준 스케줄러에 의해서 각 인덱스 값은 하나씩 증가하게 된다. 쓰레드가 다시 수행되게 되면, tail 값이 로드되지만 그 값은 이제 1이다 그 결과로 1은 tail 인덱스로 로드된다. 따라서 non-emptiness 검사는 성공하고 사용자 수준 스케줄러는 지연되어 있는 이벤트가 없더라도 수행되게 된다.

이 경우는 시스템을 비일관적인 상태로 만들지는 않고 바람직하지 못한 부하만을 초래한다 그러나 이런 경우는 거의 발생하지 않는다. 사용자 수준 스케줄러가 큐에 들어갈 때 비어있는 큐를 만나면 단순히 리턴하게 설계함으로서 이러한 구현에 따라 생길 수 있는 영향을 거의 제거하였다

3. 결 론

본 논문에서는 실시간 운영체제인 ARX의 독특한 메커니즘을 바탕으로 한 POSIX의 thread library(Pthread library)의 구현에 대하여 다루었다. ARX Pthread library는 POSIX 1003.1c의 기본적인 규약의 준수 외에 실시간 프로그래밍의 특성을 제공하기 위하여 확장된 인터페이스를 제공하였다 이와 더불어 ARX의 핵심인 가상 쓰레드와 개선된 스케쥴링 이벤트 업콜 방식을 바탕으로 사용자 수준 타이머의 관리와 사용자 수준 쓰레드와 사용자 수준 스케줄러의 동기화를 제공하는 사용자 수준 동기화 도구를 구현하였다. 이러한 Pthread library를 사용한 응용프로그램의 timing analysis의 결과는 ARX의 뛰어난 실시간 처리 능력과 함께 다양한 실시간 응용 프로그램의 요구를 수용할 수 있음을 보여준다.[2]

4. 참 고 문 헌

[1] Institute for Electrical and Electronic Engineers. POSIX Part 1: System application program interface, 1996.

[2] S.Hong, Y.Seo, and J.Park. ARX/ULTRA: A new real-time kernel architecture for supporting user-level threads. Technical Report SNU-EE-TR-1997-3, School of Electrical Engineering, Seoul National University, August 1997.

[3] F.Mueller. A library implementation of POSIX threads under UNIX. In Proceedings of 1993 USENIX Winter Technical Conference, pages 29-41, January 1993.