Check for updates

Journal of KIIT. Vol. 24, No. 10, pp. 127-136, Oct. 31, 2026. pISSN 1598-8619, eISSN 2093-7571
http://dx.doi.org/10.14801/jkiit.2026.24.10.127

멀티코어 RTOS 환경에서 공유 메모리 접근 경합 완화를 위한 Task 단위 LLQ 기반 Spinlock 기법

박준환*, 서명환**

Task-Unit LLQ-based Spinlock Techniques for Mitigating Shared Memory Access Contention in Multi-Core RTOS Environments

Junhwan Park*, Myunghwan Seo**

요 약

본 논문에서는 멀티코어 RTOS 환경에서 다수의 태스크가 공유 메모리에 접근할 때 발생하는 락 경합과 코어 간 부하 불균형을 완화하기 위해 태스크 단위 LLQ(Linked Lock Queue) 기반 스핀락 기법을 제안한다. 기존 뮤텍스 기반 방식은 동일 공유 자원에 대한 요청이 집중될 경우 태스크의 반복적인 대기·전환과 스케줄링 오버헤드로 수행 시간이 증가할 수 있다. 제안 방식은 공유 자원 접근을 태스크 노드 단위로 관리하고, 전역 큐를 통해 락 획득 순서를 제어하여 동기화 오버헤드와 코어 간 수행 시간 편차를 완화한다. Zynq UltraScale+ RFSoC 기반 Cortex-R5 듀얼 코어 FreeRTOS 환경에서 구현·비교한 결과, LLQ 방식은 뮤텍스 방식 대비 듀얼 코어 싱글 태스크 조건에서 30.84%, 다중 태스크 조건에서 평균 46.35%의 수행 시간을 개선하였으며, 코어 간 수행 시간 편차를 약 98% 감소시켰다.

Abstract

This paper proposes a task-level Linked Lock Queue (LLQ)-based spinlock technique to mitigate lock contention and inter-core load imbalance in multi-core RTOS environments. The proposed method manages shared resource access using task nodes and controls lock acquisition order through a global queue, thereby reducing synchronization overhead and execution time deviation between cores. The method was implemented and evaluated on a Cortex-R5 dual-core FreeRTOS environment based on the Zynq UltraScale+ RFSoC platform. Experimental results show that the LLQ method improved execution time by 30.84% in the dual-core single-task condition and by an average of 46.35% in the multi-task condition, while reducing inter-core execution time deviation by approximately 98%.

Keywords

RTOS, multi-core, spinlock, linked lock queue, FreeRTOS, shared memory, embedded software

* 한화시스템 연구원(교신저자)
- ORCID: https://orcid.org/0009-0004-2661-1771
** 한화시스템 수석연구원
- ORCID: https://orcid.org/0000-0002-9093-6739

· Received: Jul. 15, 2026, Revised: Aug. 26, 2026, Accepted: Aug. 29, 2026

· Corresponding Author: Junhwan Park
Pangyo R&D Center, Hanwha Systems, 305 Pangyo-ro, Bundang-gu, Seongnam-si, Gyeonggi-do, South Korea
Tel.: +82-31-8091-7196, Email: wnsghks5738@naver.com

Ⅰ. 서 론

항공우주, 국방, 의료 등 특수 목적 분야에서 사용되는 임베디드 시스템은 제한된 하드웨어 자원 내에서 높은 신뢰성과 예측 가능한 실행 시간을 요구한다. 이러한 시스템에서는 태스크의 실행 시간 보장과 결정론적 동작이 중요하므로 실시간 운영체제(RTOS, Real-Time Operating System)가 주로 사용된다[1]. 최근 임베디드 시스템의 기능이 고도화됨에 따라 기존의 단일 코어 기반 구조는 멀티코어 구조로 빠르게 전환되고 있으며[2]-[4], 이에 따라 다수의 코어와 태스크가 동일한 공유 메모리 또는 공유 자원에 접근하는 상황이 빈번하게 발생하고 있다.

멀티코어 RTOS 환경에서는 여러 태스크가 동시에 공유 자원에 접근할 수 있기 때문에 상호배제를 위한 동기화 기법이 필수적이다. 일반적으로 뮤텍스, 스핀락, 세마포어와 같은 동기화 방식이 사용되며, 이 중 스핀락은 짧은 임계구역에 대해 빠르게 락을 획득하고 해제할 수 있다는 장점이 있다[5]. 그러나 공유 변수 기반의 기존 락 방식은 다수의 태스크가 동일한 락 변수에 반복적으로 접근할 경우 메모리 버스 트래픽과 원자 연산 경합을 증가시킬 수 있다. 특히 멀티코어 환경에서는 특정 코어 또는 특정 태스크에 락 획득 기회가 집중되어 코어 간 수행 시간 편차가 발생할 수 있으며, 이는 RTOS 환경에서 중요한 공정성 및 시간 결정성을 저하시킬 수 있다[6].

이러한 문제를 완화하기 위해 MCS Lock[7]과 같은 Queue-based Locking 기법이 연구되어 왔다. MCS Lock은 락을 요청하는 코어나 스레드를 연결 리스트 형태의 큐로 관리하여 공유 변수에 대한 반복 접근을 줄이고, 높은 경합 상황에서도 확장성을 제공한다[8]. 또한 큐 기반 락 기법은 주로 범용 운영체제나 일반 멀티프로세서 환경을 대상으로 설계되었으며, RTOS의 태스크 관리 구조와 직접적으로 연계된 방식은 제한적으로 다루어졌다. RTOS 환경에서는 코어 단위뿐 아니라 태스크 단위의 실행 순서, 대기 상태, 우선순위, 수행 시간 예측 가능성이 중요하므로, 기존의 코어 또는 스레드 중심 락 구조를 그대로 적용하는 데에는 한계가 있다[9]. 멀티코어 RTOS 환경의 spinlock 연구로, Nagarajan et al.[10]은 AUTOSAR[11] 기반 멀티코어 실시간 운영체제에서 공정성을 제공하고 기아 현상(Starvation)을 해결하기 위한 M-HLP Spinlock Protocol을 제안하였다. 해당 연구는 Ticket System을 이용하여 FIFO 순서를 보장하고, 락 획득 시 태스크 우선순위를 가장 높게 조정함으로써 차단 시간(Blocking time)의 상한을 분석할 수 있도록 한 점이 특징이다. 그러나 M-HLP는 각 코어의 태스트 우선순위와 스케줄러 상태를 OS 수준에서 일관되게 관리할 수 있는 AUTOSAR 멀티코어 RTOS 모델을 전제로 하므로, 코어별 OS 또는 스케줄러가 독립적으로 동작하는 AMP(Asymmetric Multi-Processing) 환경에서는 그대로 적용하기 어렵다. 또한 대기 중인 태스크들이 공통 Now-Serving Counter를 반복적으로 확인하는 Ticket 기반 구조이므로, 경합이 증가할 경우 특정 공유 변수에 대한 Polling 접근이 집중될 수 있다.

본 논문에서는 AMP 기반 멀티코어 RTOS 환경에서도 적용 가능한 태스크 단위 LLQ(Linked Lock Queue) 기반 스핀락 기법을 제안한다. 제안하는 LLQ 방식은 공유 자원에 접근하는 각 태스크를 개별 노드로 구성하고, 전역 큐를 통해 락 획득 순서를 관리한다. 이를 통해 동일 공유 자원에 대한 락 요청을 하나의 공유 변수에 집중시키는 기존 뮤텍스 기반 방식과 달리, 태스크별 대기 구조를 형성하여 경합을 분산시키고 락 획득 순서를 명확하게 제어할 수 있다. 또한 각 코어의 스케줄러 통합 제어나 priority ceiling 처리에 의존하지 않고 공유 메모리와 원자 연산을 기반으로 큐 상태를 관리하므로, 코어별 RTOS가 독립적으로 동작하는 AMP 구조에서도 적용 가능하다.

첫째, 멀티코어 FreeRTOS 환경에서 공유 메모리 접근 경합을 완화하기 위한 태스크 단위 LLQ 기반 스핀락 구조를 설계하였다. 둘째, 공유 메모리 내 태스크 노드와 전역 Queue Tail을 이용하여 락 획득 및 해제 순서를 관리하는 알고리즘을 구현하였다. 셋째, Zynq UltraScale+ RFSoC 기반 Cortex-R5 듀얼 코어, AMP 환경에서 기존 뮤텍스 기반 방식과 LLQ 기반 방식을 비교하여 수행 시간, 코어 간 편차, 공정성 측면의 개선 효과를 분석하였다..

본 논문의 구성은 다음과 같다. II장에서는 기존 동기화 기법과 큐 기반 락 구조를 검토하고, III장에서는 제안하는 LLQ 기반 스핀락 구조와 동작 알고리즘을 설명한다. IV장에서는 실험 환경, 측정 방법, 성능 평가 결과를 제시하고, V장에서 결론 및 향후 연구 방향을 정리한다.

II. 관련 연구

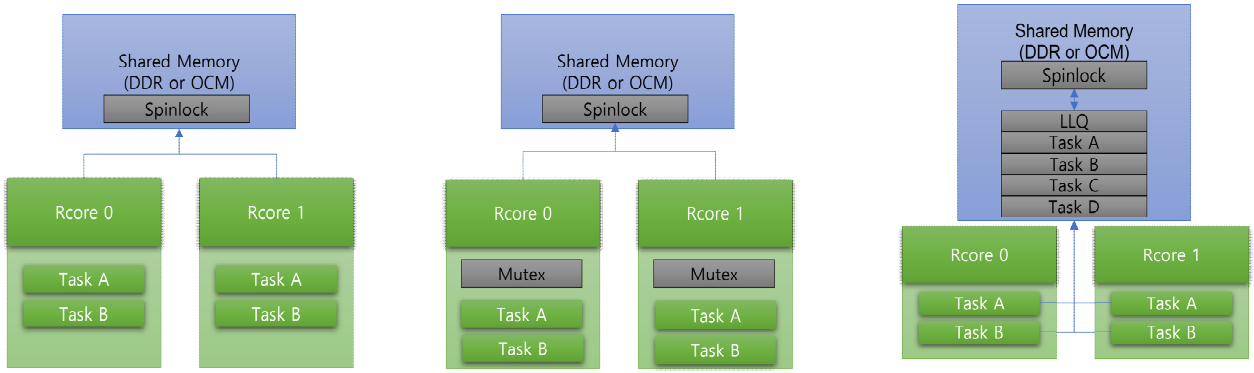
멀티코어 시스템에서 공유 자원 접근을 제어하기 위한 대표적인 동기화 기법으로 뮤텍스와 스핀락이 사용된다. 뮤텍스는 특정 태스크가 공유 자원을 점유하는 동안 다른 태스크의 접근을 차단하여 상호 배제를 보장한다. 구현이 단순하고 일반적인 운영체제 환경에서 널리 사용되지만, 락 경쟁이 증가할 경우 태스크 대기 시간이 길어지고 문맥 전환(Context switching) 와 스케줄링 오버헤드가 발생할 수 있다.

스핀락은 락을 획득할 수 있을 때까지 태스크 또는 코어가 반복적으로 락 변수를 확인하는 방식이다. 임계구역이 짧고 락 보유 시간이 매우 짧은 경우에는 효율적일 수 있지만, 다수의 코어가 동일한 락 변수에 반복적으로 접근하면 공유 메모리 접근 경합과 원자 연산 부하가 증가한다. 특히 멀티코어 환경에서는 각 코어가 동일 락 변수에 접근하면서 메모리 버스 트래픽이 증가하고, 특정 코어에 락 획득 기회가 집중될 경우 코어 간 부하 불균형이 발생할 수 있다.

이러한 문제를 해결하기 위해 Queue-based Locking 기법이 제안되었다. 대표적인 방식인 MCS Lock은 락 요청자를 연결 리스트 형태의 큐로 관리하여 각 요청자가 자신의 노드에서 대기하도록 한다. 이를 통해 모든 요청자가 동일 공유 변수에 반복 접근하는 문제를 줄이고, 락 획득 순서를 보장할 수 있다. MCS Lock은 높은 경합 상황에서 확장성과 공정성 측면의 장점을 가진다.

그러나 기존 큐 기반 락 기법은 주로 범용 운영체제의 스레드 또는 멀티프로세서의 코어 단위 동기화에 초점을 맞추고 있다. 반면 RTOS 환경에서는 태스크 단위의 실행 제어가 중요하며, 태스크의 대기 상태, 실행 순서, 우선순위, Worst-case 지연 시간 등이 시스템의 실시간성을 결정하는 핵심 요소가 된다. 따라서 멀티코어 RTOS 환경에서는 기존 큐 기반 락 구조를 태스크 단위로 확장하고, RTOS의 태스크 실행 구조와 연계할 수 있는 동기화 방식이 필요하다.

본 논문에서 제안하는 LLQ 기반 스핀락은 기존 Queue-based Locking의 장점을 유지하면서도, 락 관리 단위를 RTOS 태스크 수준으로 확장한다는 점에서 차별성을 가진다. 즉, 기존 방식이 주로 코어 또는 스레드 단위의 락 대기 구조를 형성하는 데 초점을 두었다면, 제안 방식은 그림 1와 같이 공유 자원에 접근하는 개별 태스크를 노드로 관리하고, 전역 큐를 통해 태스크 간 락 획득 순서를 직접 제어한다. 이를 통해 멀티코어 RTOS 환경에서 태스크 간 공정성과 실행 시간 예측 가능성을 높이는 것을 목표로 한다.



(a) 스핀락 구조
(a) Spinlock structure

(b) 뮤텍스 기반 스핀락 구조
(b) Mutex based Spinlock structure

(c) LLQ 기반 스핀락 구조
(c) LLQ based Spinlock structure

그림 1. 스핀락 의 구조 비교
Fig. 1. Comparison of the structure of spinlock

III. 제안하는 태스크 단위 LLQ 기반 스핀락

제안하는 LLQ 기반 스핀락의 설계 목표는 멀티코어 RTOS 환경에서 공유 메모리 접근 시 발생하는 락 경합을 완화하고, 태스크 간 락 획득 순서를 명확하게 관리하는 것이다.

기존 뮤텍스 기반 방식은 공유 자원에 접근하는 모든 태스크가 동일한 락 변수에 접근하기 때문에, 경합이 증가할수록 문맥 전환(Context switching)과 스케줄링 오버헤드가 집중되어 코어 간 수행 시간 편차가 커질 수 있다. 반면 LLQ 방식은 각 태스크를 개별 노드로 관리하고, 전역 큐를 통해 락 획득 순서를 제어함으로써 공유 변수 접근 집중을 줄이고 태스크간 수행시간 예측가능성 향상 및 대기 순서를 보장한다.

3.1 Linked lock queue 구조

제안하는 LLQ 구조는 알고리즘 1와 같이 크게 태스크 노드와 Global Lock Queue로 구성된다. 태스크 노드는 공유 자원 접근을 요청하는 개별 태스크의 정보를 저장하는 구조체이다. 태스크 노드에는 태스크 ID, 태스크 상태, 다음 노드를 가리키는 포인터, 락 획득 여부를 나타내는 플래그가 포함된다. 각 태스크는 공유 자원 접근을 요청할 때 자신의 태스크 노드를 초기화하고, 전역 큐에 등록한다. Global Lock Queue는 공유 자원에 접근하려는 전체 태스크의 대기 순서를 관리하는 전역 구조이다. 이 구조는 큐의 마지막 노드를 가리키는 Tail 정보와 현재 공유 자원을 점유하고 있는 태스크 ID를 포함한다. Tail 정보는 새로운 태스크가 큐에 진입할 때 원자 연산을 통해 갱신되며, 현재 점유 태스크 정보는 공유 자원의 소유권을 확인하는 데 사용된다.

이러한 구조를 통해 LLQ 방식은 모든 태스크가 하나의 공유 락 변수에 반복적으로 접근하는 대신, 최초 큐 등록 이후 각 태스크가 자신의 노드 상태를 기준으로 대기하도록 한다. 따라서 경합이 높은 환경에서도 공유 변수 접근 집중을 완화할 수 있다.

알고리즘 1. LLQ 구조와 lock 획득 및 해제 알고리즘
Algorithm 1. LLQ structure and lock acquisition and release algorithm

```
typedef struct:: task_node
 task_id            // 태스크 ID
 task_state         // 태스크 상태
 task_node *next_task       // next 획득 태스크
 lock_acquire     // 락 획득 플래그

typedef struct:: llq_lock_t
 queue_tail  // LLQ 의 마지막 노드
 owner_task_id  // 현재 락 소유 태스크 ID

function llq_lock_acquire ( llq_lock_t *lock , task_node *node )
  node->next = null
  atomic_flag_clear_explicit(
            node->lock_acquire, memory_order_release)
  node->state = READY
  task_node *prev = atomic_exchange_explicit(
            lock->queue_tail, node, memory_order_acq_rel)
  if(prev != null)
            atomic_store_explicit(prev->next, node,
memory_order_acquire)
            node->state = WAITING
            while(atomic_flag_test_and_set_explicit(
                        node->lock_acquire,
memory_order_aucqire)
  else
            lock->owner_task_id = node->task_id
            node->state = RUNNING

function llq_lock_release ( llq_lock_t *lock , task_node
*node)
  task_node *next = atomic_load_explicit(node->next,
memory_order_acquire)
  if(!next)
            task_node *expect = node
            if(atomic_compare_exchange_explicit(
                        lock->queue_tail, expect, null,
                        memory_order_acq_rel,
memory_order_acquire)
                        lock->owner_task_id = -1
                        return
            while(!(next = atomic_load_explicit(node->next,
                                                  /
memory_order_release)))
  lock->owner_task_id = next->task_id
  next->state = RUNNING
  atomic_flag_clear_explicit(next->lock_acquire,
memory_order_release)
```

3.2 Lock Acquire 알고리즘

LLQ 기반 락 획득 절차는 그림 2와 같다.

첫째, 공유 자원 접근을 요청하는 태스크는 자신의 태스크 노드를 초기화한다. 이때 다음 노드 포인

터는 NULL로 설정하고, 락 획득 플래그는 비활성 상태로 초기화한다. 둘째, 태스크는 Atomic Exchange 연산을 이용하여 Global Lock Queue의 Tail을 자신의 노드로 변경한다. 이 과정에서 기존 Tail 정보를 반환받는다. 셋째, 기존 Tail 정보가 NULL인 경우, 현재 태스크는 큐의 첫 번째 요청자이므로 즉시 락을 획득한다. 이때 현재 공유 자원 소유자 정보에 자신의 태스크 ID를 기록하고 실행 상태로 전환한다. 넷째, 기존 Tail 정보가 존재하는 경우, 현재 태스크는 기존 Tail 노드의 다음 노드로 연결된다. 이후 자신의 락 획득 플래그가 활성화될 때까지 대기 상태를 유지한다. 이전 태스크가 공유 자원을 해제하면 현재 태스크의 락 획득 플래그가 활성화되고, 해당 태스크는 공유 자원에 접근할 수 있다.

이 방식은 락 획득 순서를 큐에 등록된 순서대로 관리하므로, 특정 태스크가 반복적으로 락을 선점하는 상황을 회피할 수 있다.

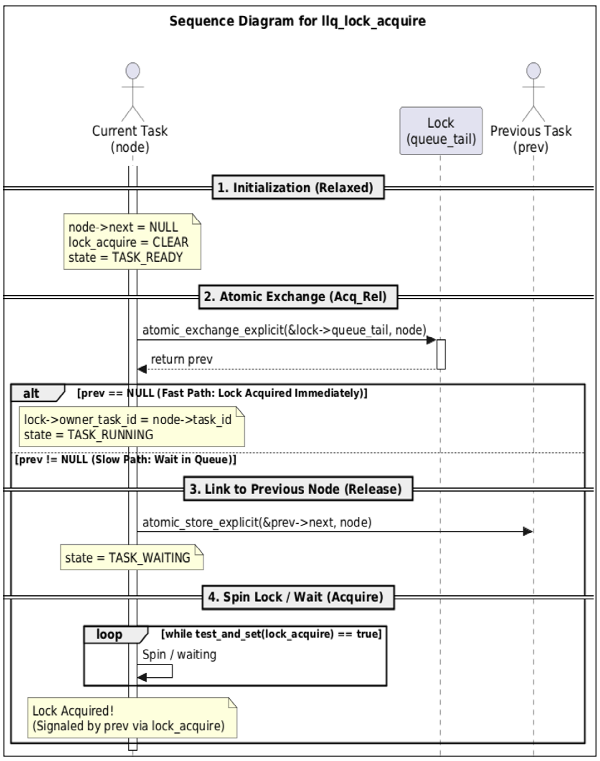


그림 2. LLQ 스핀락 획득 알고리즘에 대한 시퀀스 다이어그램

Fig. 2. sequence diagram for LLQ spin-lock acquisition algorithm

3.3 Lock Release 알고리즘

LLQ 기반 락 해제 절차는 그림 3와 같다.

첫째, 현재 공유 자원을 점유하고 있는 태스크는 자신의 태스크 노드에서 다음 노드 정보를 확인한다. 둘째, 다음 노드가 존재하는 경우, 현재 태스크는 공유 자원의 소유자 정보를 다음 태스크 ID로 갱신하고, 다음 태스크의 락 획득 플래그를 활성화한다. 이를 통해 다음 태스크가 공유 자원에 접근할 수 있도록 한다. 셋째, 다음 노드가 존재하지 않는 경우, 현재 태스크는 Atomic Compare Exchange 연산을 이용하여 Global Lock Queue의 Tail이 자신의 노드와 일치하는지 확인한다. 두 정보가 일치하면 큐에 대기 중인 태스크가 없는 것으로 판단하고, Tail을 NULL로 변경한 뒤 공유 자원을 미점유 상태로 전환한다. 넷째, Tail 정보가 자신의 노드와 일치하지 않는 경우에는 새로운 태스크가 큐에 등록되는 중간 상태로 판단한다. 이 경우 현재 태스크는 다음 노드 정보가 등록될 때까지 대기한 후, 해당 태스크에 락 획득 권한을 전달한다.

이와 같은 획득 및 해제 절차를 통해 LLQ 방식은 공유 자원 접근 순서를 명확히 관리하며, 대기 중인 태스크가 순차적으로 락을 획득하도록 보장한다.

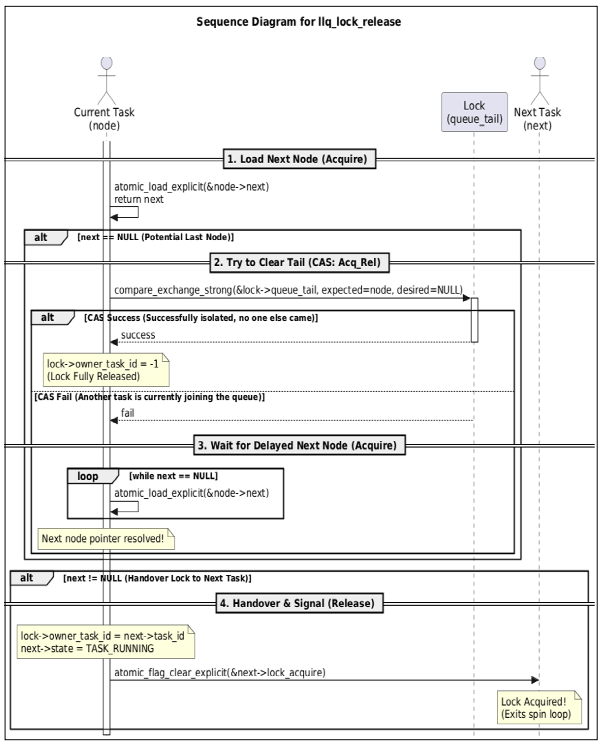


그림 3. LLQ 스핀락 해제 알고리즘에 대한 시퀀스 다이어그램

Fig. 3. sequence diagram for LLQ spin-lock release algorithm

3.4 RTOS 우선순위 스케줄링과 LLQ 실행순서

제안하는 LLQ 기반 스핀락 방식은 RTOS 의 태스크 우선순위 스케줄링 정책과 독립적으로 동작하도록 설계되었다. 공유 자원 접근을 요구하는 태스크는 우선순위와 무관하게 대기 큐에 FIFO 순서로 등록됨으로써 자원 선점 순서가 결정되므로, 우선순위가 높은 태스크가 다른 태스크들의 대기 중에 자원을 요구하더라도 RTOS 스케줄러에 의한 우선순위 역전 현상은 발생하지 않는다. 구체적으로, 자원을 선점 중인 태스크보다 우선순위가 높은 태스크가 락 획득을 시도할 경우 기존 태스크는 스케줄러에 의해 일시적으로 동작이 중단될 수 있으나, 락 소유권은 기존 태스크에게 유지된다. 따라서 고 우선순위 태스크는 자원을 즉시 선점하는 것이 아니라 공유 자원 접근을 위한 FIFO 대기 큐에 등록되어 대기 상태가 되며, 기존 태스크는 자원 선점 상태를 유지하며 임계 영역을 계속 수행한다. 이때 대기 큐에 등록된 순서 또한 변경되지 않으므로, 우선순위 역전 없이 자원 접근 순서가 보장된다.

3.5 기존 방식과의 차이

기존 뮤텍스 기반 방식은 공유 자원 접근 전 OS 스케줄링에 의한 뮤텍스 락을 통해 하나의 태스크만 단일 락 변수를 통해 공유 메모리에 접근한다. 이 방식은 구현이 단순하다는 장점이 있지만, 다수의 태스크가 동시에 공유 자원에 접근할 경우 커널 모드 전환에 의한 문맥 전환(Context switching) 빈도가 증가하고, 멀티코어 간 스케줄링 오버헤드가 증가한다. 결과적으로 특정 코어 또는 특정 태스크의 락 획득이 지연되어 코어간 시간편차가 증가할수 있다.

반면 제안하는 LLQ 방식은 각 태스크를 독립적인 노드로 관리하고, 전역 큐를 통해 태스크의 락 획득 순서를 엄격히 제어한다. 각 태스크는 큐 등록 이후 자신의 노드 상태를 기준으로 스핀 대기하므로, 동일 공유 변수에 대한 불필요한 스케줄러 호출 및 반복 접근을 줄일 수 있다. 태스크 간 공정성을 확보하고 기아 현상(Starvation) 과 우선순위 역전 현상 가능성을 줄일 수 있다.

따라서 제안 방식은 기존 뮤텍스 기반 방식과 비교하여 다음과 같은 차별성을 가진다. 첫째, 락 관리 단위가 코어 또는 단일 공유 변수 중심이 아닌 태스크 단위의 실행 순서, 대기 상태, 우선순위, 수행 시간 예측 가능성을 고려하기 위한 RTOS 태스크 단위이다. 둘째, 공유 자원 접근 순서를 전역 큐를 통해 명시적으로 관리하여 락 경쟁 시 발생하는 경합을 최소화한다. 셋째, 멀티코어 환경에서 코어간 수행 시간 편차를 줄여 각 태스크에서의 실행 시간 성능이 일정하므로 분석의 정확도를 높이고 실행 시간 예측 가능성을 높일 수 있다.

IV. 실험 환경 및 성능 평가

4.1 실험 환경

제안하는 LLQ 기반 스핀락의 성능을 평가하기 위해 Xilinx Zynq UltraScale+ RFSoC 기반 실험 환경을 그림 4와 같이 구성하였다. 시스템은 Cortex-A53 코어와 Cortex-R5 코어로 구성되며, Cortex-A53에서는 PetaLinux 기반 시스템을 구동하고 Cortex-R5에서는 FreeRTOS 기반 실시간 태스크를 실행하였다.

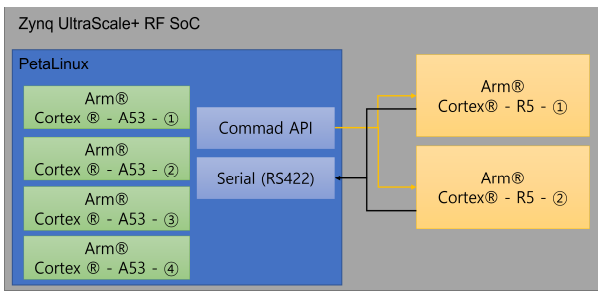


그림 4. Zynq UltraScale+ RF SoC OpenAMP 실험환경
Fig. 4. Zynq Ultrascale+ RF SoC experimental environment

A53 코어는 OpenAMP 환경을 통해 R5 코어의 펌웨어 로드, 실행 시작, 정지 명령을 수행하였다. 두 개의 R5 코어는 각각 독립적인 FreeRTOS 스케줄러를 실행하며, 실험 조건에 따라 코어당 1개 또는 4개의 태스크를 실행하였다. 공유 자원은 두 R5 코어가 접근 가능한 DDR 메모리 영역에 구성하였다. 캐시 일관성 문제로 인한 데이터 불일치를 방지

하기 위해 공유 메모리 영역은 non-cacheable 영역으로 설정하였다. 또한 실험 결과 로그는 DDR 메모리 내 별도 trace 영역에 저장하고, A53 코어에서 확인하였다.

4.2 측정 방법

본 실험에서는 기존 뮤텍스 기반 방식과 제안 LLQ 기반 방식을 동일한 하드웨어 및 소프트웨어 환경에서 비교하였다.

공유 메모리 접근 가능 용량은 메모리 용량 제한으로 인해 204,800 byte로 제한된다. 단일 태스크 환경에서는 이 메모리를 512 byte씩 분할하여 접근하며, 이에 따라 총 400 회의 공유 메모리 락 요청이 발생한다. 다중 태스크 조건에서는 각 코어에서 4개의 태스크가 동시에 공유 자원에 접근하도록 구성하였으며, 각 태스크는 동일한 횟수의 공유 메모리 접근을 수행하였다. 실험은 단일 코어 환경과 듀얼 코어 환경으로 구분하여 수행하였다.

본 논문에서는 성능 평가를 위해 다음 세 가지 지표를 사용하였다.

첫째, 평균 수행 시간은 락 요청부터 공유 자원 접근 및 락 해제까지 소요된 평균 시간을 의미하며, 5회 반복 측정 결과를 평균하였다. 둘째, 성능 개선율은 뮤텍스 기반 방식 대비 LLQ 기반 방식의 수행 시간 감소 비율로 계산한다. 셋째, 코어 간 수행 시간 편차는 Rcore0과 Rcore1의 수행 시간 차이로 정의하며, 이 값이 작을수록 두 코어 간 자원 획득 기회가 균등하게 분배된 것으로 해석한다.

성능 개선율 R_I (%)는 다음과 같이 계산하였다.

$$R_I = (T_m - T_l)/T_m \times 100 \quad (1)$$

식 (1)에서 T_m은 뮤텍스 수행시간을 의미하고, T_l은 LLQ 수행시간을 의미한다.

코어 간 수행 시간 편차 $\delta_{t.core}$는 다음과 같이 정의하였다.

$$\delta_{t.core} = | T_{R.0} - T_{R.1} | \quad (2)$$

식 (2)에서 $T_{R.0}$는 Rcore 0에서의 수행시간을 의미하며, $T_{R.1}$은 Rcore 1에서의 수행시간을 의미한다.

4.3 단일 코어 환경에서의 성능 비교

그림 5의 Single-1Task 와 Single-4Task 는 단일 코어 환경에서 두 방식의 기본 수행 성능을 비교한 결과이다. 단일 태스크가 512 byte 크기의 공유 메모리에 400 회 접근하는 조건에서 뮤텍스 기반 방식은 20.923 μs, LLQ 기반 방식은 13.282 μs로 측정되었다. 두 방식 간 차이는 약 36.519%로 나타났다.

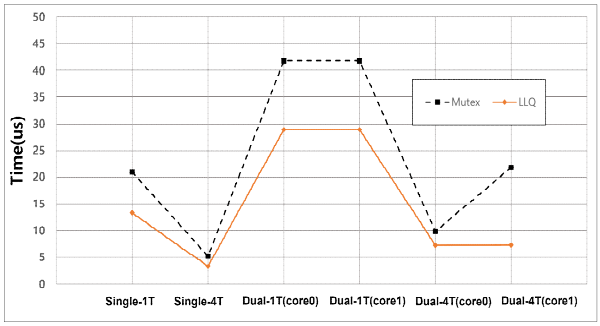


그림 5. 단일 및 듀얼 코어 환경에서의 뮤텍스 기반 스핀락 과 LLQ 기반 스핀락 의 전체 성능 비교
Fig. 5. Overall performance comparison of Mutex-based spinlock and LLQ-based spinlock in single and dual-core environments

또한 단일 코어에서 4개의 태스크가 각각 공유 자원 접근을 수행하는 경우, 뮤텍스 기반 방식은 평균 5.265 μs, LLQ 기반 방식은 평균 3.346 μs로 측정되었다. 이 경우 성능 차이는 약 36.380% 수준으로 나타났다.

이는 단일 코어 환경에서도 공유 자원 경합이 발생하기 때문에 락 구조의 차이가 전체 수행 시간에 미치는 영향이 있음을 의미한다. 구체적으로 뮤텍스 방식은 락 획득 실패 시 태스크 블록 및 스케줄러 호출이 빈번히 발생하여 Rore 최대 clock Frequency 기준으로 약 533 MHz 마다 문맥 전환(Context switching) 오버헤드가 누적되는 반면, LLQ 방식은 큐 기반 대기 구조를 통해 불필요한 스케줄러 개입을 최소화했기 때문이다. 즉, 단일 코어 환경에서도 태스크 단위의 락 획득 구조는 스케줄러 활성화 횟수 감소를 통해 수행 시간에 영향을 미치는 것으로 볼 수 있다.

4.4 듀얼 코어 환경에서의 성능 비교

듀얼 코어 환경에서는 두 코어가 동일한 공유 자원에 접근하므로 락 획득 경쟁이 발생한다. 각 코어에서 1개의 태스크가 400회 공유 메모리 접근을 수행한 결과, 그림 5의 코어 0과 코어 1에 대한 Dual-1Task와 Dual-4Task의 뮤텍스 기반 방식은 평균 41.795 μs, LLQ 기반 방식은 평균 28.905μs로 측정되었다. 이는 LLQ 방식이 뮤텍스 방식 대비 약 30.84%의 성능 개선을 보였음을 의미한다. 뮤텍스 기반 방식에서는 공유 자원 접근으로 인한 경합 시간 보다 스핀락 대기 중 발생하는 경합 및 문맥교환 (Context switching) 및 스케줄링 오버헤드로 인한 성능 저하가 발생한다. 반면 LLQ 방식 은 락 요청 태스크를 큐에 등록한 후 각 태스크가 자신의 노드 상태를 기준으로 대기하므로, 공유 락 변수의 원자적 연산 (Atomic operation) 횟수를 최소화하여 공유 락 변수에 대한 반복 접근이 감소한다. 이러한 구조적 차이로 인해 듀얼 코어 환경에서 LLQ 방식의 수행 시간이 더 낮게 나타난다.

4.5 태스크 수 증가에 따른 성능 영향

그림 6과 그림 7은 코어당 4개의 태스크가 동시에 공유 메모리에 접근하는 고경합 환경에서 두 방식의 성능을 추가로 비교한 결과이다. 뮤텍스 기반 방식의 경우 Rcore0에서는 9.822μs, Rcore1에서는 21.832μs로 측정되었다.

두 코어 간 수행 시간 차이는 약 12.009 μs로 나타났으며, 이는 스케줄러의 선점 우선순위 경쟁으로 인해 특정 코어에 락 획득 요청이 집중되어 코어 간 부하 불균형이 발생했음을 의미한다.

반면 LLQ 기반 방식은 Rcore0에서 7.263μs, Rcore1에서 7.276 μs로 측정되었다. 두 코어 간 수행 시간 차이는 약 0.0129 μs로, 뮤텍스 기반 방식에 비해 현저히 작았다. 성능 개선율은 Rcore0 기준 약 26.057%, Rcore1 기준 약 66.67%로 나타났다.

이 결과는 태스크 수가 증가하고 공유 자원 경합이 심화될수록 LLQ 방식의 장점이 더욱 뚜렷해짐을 보여준다. LLQ 방식은 전체 태스크를 전역 큐로 관리하여 공유 자원 접근 순서를 순차적으로 제어하므로, 특정 코어 또는 특정 태스크에 락 획득 기회가 집중되는 현상을 완화할 수 있다. 이는 큐 기반 메커니즘이 코어 간 락 경쟁을 분산시켜 대기 시간으로 인한 성능 하락을 줄였기 때문이다.

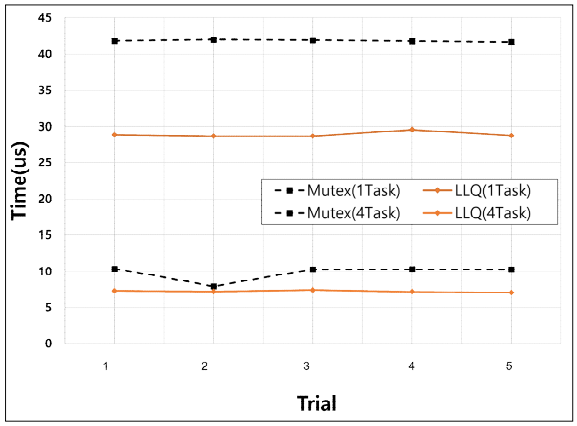


그림 6. 듀얼 코어 환경에서 코어0 에 대한 Muutex 및 LLQ 기반 스핀락의 코어 간 수행 시간 편차 비교
Fig. 6. Comparison of running time deviations between cores of Muutex and LLQ-based spinlock for core0 in dual-core environments

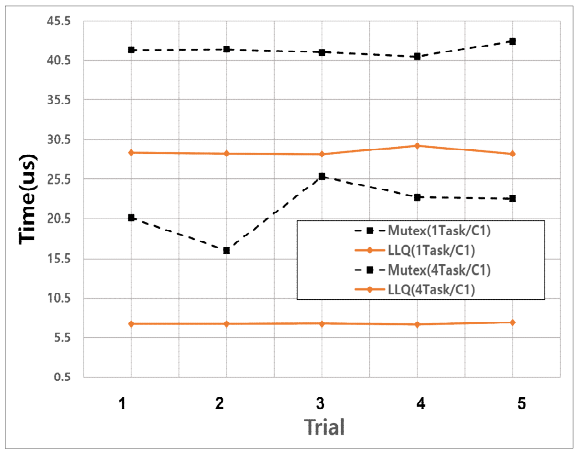


그림 7. 듀얼 코어 환경에서 코어1에 대한 Muutex 및 LLQ 기반 스핀락의 코어 간 수행 시간 편차 비교
Fig. 7. Comparison of running time deviations between cores of Muutex and LLQ-based spinlock for core1 in dual-core environments

4.6 공정성 및 시간 결정성 분석

본 논문에서는 코어 간 수행 시간 편차를 공정성 평가 지표로 정의하였다. 그림 8와 같이 뮤텍스 기반 방식의 경우 코어당 4개의 태스크가 동시에 공유 자원에 접근하는 조건에서 Rcore0과 Rcore1의 수

행 시간 차이는 최대 약 15.56 μs로 나타났다. 반면 LLQ 기반 방식에서는 두 코어 간 수행 시간 차이가 약 0.318 μs에 불과하였다.

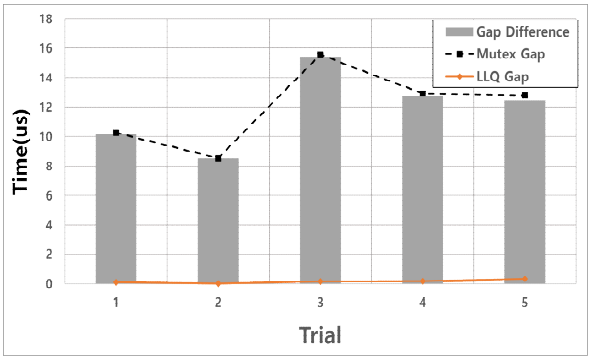


그림 8. 코어 간 성능 편차 그래프
Fig. 8. Performance deviation graph between cores

이는 LLQ 방식이 공유 자원 접근 순서를 전역 큐를 통해 관리함으로써 두 코어 간 락 획득 기회를 보다 균등하게 분배했음을 의미한다. 뮤텍스 기반 방식은 각 코어에서 독립적으로 락 획득 경쟁이 발생하므로 특정 코어가 상대적으로 더 긴 대기 시간을 가질 수 있다. 반면 LLQ 방식은 모든 태스크를 하나의 큐 구조에서 관리하기 때문에 락 획득 순서가 명확하며, 태스크 간 기아 현상(Starvation) 가능성을 줄일 수 있다.

또한 반복 실험에서도 LLQ 방식은 수행 시간 편차가 작게 유지되는 경향을 보였다. 이는 LLQ 방식이 단순히 평균 수행 시간을 줄이는 것뿐 아니라, 실행 시간의 변동성을 줄여 RTOS 환경에서 중요한 시간 결정성 향상에도 기여할 수 있음을 의미한다. 특히 항공우주, 국방, 의료 등 실시간성이 중요한 임베디드 시스템에서는 평균 성능뿐 아니라 Worst-case 지연 시간과 실행 시간 예측 가능성이 중요하므로, LLQ 기반 구조는 실시간 시스템의 동기화 기법으로 활용 가능성이 있다.

V. 결 론

본 논문에서는 멀티코어 RTOS 환경에서 공유 메모리 접근 시 발생하는 락 경합 문제를 완화하기 위해 태스크 단위 Linked Lock Queue 기반 스핀락 기법을 제안하였다. 기존 큐 기반 락 기법은 주로 범용 운영체제나 일반 멀티프로세서 환경을 대상으로 설계되었다. 본 연구에서 제안하는 방식은 공유 자원에 접근하는 각 태스크를 개별 노드로 구성하고, 전역 큐를 통해 락 획득 순서를 관리함으로써 기존 뮤텍스 기반 방식에서 발생할 수 있는 공유 변수 접근 집중과 코어 간 수행 시간 불균형 문제를 효과적으로 완화한다.

제안 기법의 성능을 실 측정 결과 기반으로 분석하기 위해 다중 코어 환경을 구성하여 실험하였다. 이를위해 Xilinx Zynq UltraScale+ RFSoC 환경에서 Cortex-R5 듀얼 코어에 FreeRTOS 를 적용하고, 일반적으로 사용되는 기존 동기화 기법인 뮤텍스 기반 방식과 LLQ 기반 방식을 비교하였다. 실험 결과, 단일 코어 환경과 듀얼 코어 환경에서 LLQ 기반 방식이 뮤텍스 기반 방식 대비 약 36.380% 향상된 수행 시간 개선을 보였다. 또한 코어당 4 개의 태스크가 동시에 공유 자원에 접근하는 고경합 환경에서는 코어별로 약 26.057% 및 66.672% 의 성능 개선이 확인되었으며, 코어 간 수행 시간 편차도 크게 감소하였다. 이러한 결과는 제안하는 LLQ 기반 스핀락 방식이 단순한 수행 시간 개선뿐 아니라, 멀티코어 RTOS 환경에서 태스크 간 공정성과 실행 시간 예측 가능성을 향상시키는 데 효과적임을 보여준다. 특히 공유 자원 접근 경합이 빈번하게 발생하는 실시간 임베디드 시스템에서 LLQ 방식은 기존 동기화 기법들을 보완할 수 있는 대안으로 활용될 수 있다.

다만 본 연구는 Cortex-R5 듀얼 코어 환경과 제한된 태스크 수 조건에서 수행되었으며, 태스크 우선순위가 서로 다른 환경이나 캐시 활성화 조건에서의 성능 영향은 추가적으로 검토할 필요가 있다.

향후 연구에서는 태스크 우선순위 차이에 따른 스케줄링 영향과 캐시 활성화 조건에서의 성능 변화를 심층 분석할 계획이다. 구체적으로는 우선순위를 반영한 Priority-aware LLQ 구조를 설계하여 고경합 환경에서의 실시간 성 보장 능력을 강화할 것이다. 아울러 코어 수를 확장한 많은 코어(Many-core) 환경과 다양한 하드웨어 아키텍처에서의 검증을 수행하여 제안 기법의 확장성과 일반성을 입증하고자 한다.

References

[1] H. Jang, S. Park, and H. Kim, "Deferrable Task Execution Model for Reducing Memory Interference in a Real-Time Multi-Core Embedded System", The Journal of Korean Institute of Information Technology, Vol. 22, No. 2, pp. 63-69, Feb. 2024. https://doi.org/10.14801/jkiit.2024.22.2.63.

[2] S. Saidi, R. Ernst, S. Uhrig, H. Theiling, and B. D. Dinechin, "The shift to multicores in real-time and safety-critical systems", Proc. 2015 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS), Amsterdam, Netherlands, pp. 220-229, Oct. 2015. https://doi.org/10.1109/CODESISSS.2015.7331385.

[3] R. Medina, E. Borde, and L. Pautet, "Generalized Mixed-Criticality Static Scheduling for Periodic Directed Acyclic Graphs on Multi-Core Processors", IEEE Trans. on Computers, Vol. 70, No. 3, pp. 457-470, Apr. 2020. https://doi.org/10.1109/TC.2020.2990229.

[4] M. Bottaro and T. Vardanega, "Evaluating a multicore Mixed-Criticality System implementation against a temporal isolation kernel", Journal of Systems Architecture, Vol. 130, pp. 1-14, Sep. 2022. https://doi.org/10.1016/j.sysarc.2022.102688.

[5] ARM, ARM Architecture Reference Manual ARMv8, 2022.

[6] T. E. Anderson, "The performance of spin lock alternatives for shared-memory multiprocessors", IEEE Transactions on Parallel and Distributed Systems, Vol. 1, No. 1, pp. 6-16, Jan. 1990.

[7] J. M. Mellor-Crummey and M. L. Scott, "Algorithms for scalable synchronization on shared-memory multiprocessors", ACM Transactions on Computer Systems, Vol. 9, No. 1, pp. 21-65, Feb. 1991.

[8] T. David, J. Guerraoui, and V. Trigonakis, "Everything you always wanted to know about synchronization but were afraid to ask", Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP), Farminton Pennsylvania, pp. 33-48, Nov. 2013. https://doi.org/10.1145/2517349.2522714.

[9] B. Brandenburg and J. Anderson, "Spin-based reader-writer synchronization for multiprocessor real-time systems", Real-Time Systems, Vol. 46, No. 1, pp. 25-87, Sep. 2010. https://doi.org/10.1007/s11241-010-9097-2.

[10] D. Nagarajan, T. Scheipel, and M. Baunach, "Fair and starvation-free spin-lock for real-time autosar systems: M-hlp", Proceedings of the 39th ACM/SIGAPP Symposium On Applied Computing, Ávila, Spain, pp. 437-445, Apr. 2024. https://doi.org/10.1145/3605098.3635922.

[11] AUTOSAR, AUTOSAR Specification of Operating System, https://bit.ly/3kFJ4SC. [accessed: Jan. 15, 2023]

저자소개

박 준 환 (Junhwan Park)



2022년 2월 : 한국공학대학교 전파공학과(공학사)
2023년 4월 ~ 현재 : ㈜한화시스템 연구원
관심분야 : 임베디드 소프트웨어

서 명 환 (Myunghwan Seo)



2002년 2월 : 충남대학교 정보통신공학과(학사)
2004년 2월 : 한국과학기술원 전자공학과(석사)
2009년 2월 : 한국과학기술원 전자공학과(박사)
2009년 1월 ~ 현재 : ㈜한화시스템 수석연구원
관심분야 : Mobile Ad-hoc Network, Network Synchronization, Military Wireless Communication, 임베디드 소프트웨어