Smart Media Journal / Vol.14, No.12 / ISSN:2287-1322
https://dx.doi.org/10.30693/SMJ.2025.14.12.28

RTEMS SMP 지원 현황 및 스케줄링 기법 분석

(Analysis of RTEMS SMP Support Status and Scheduling Techniques)

박주찬*, 김권혁, 박성민***, 허금숙***, 장준혁******

(Juchan Park, Gwonhyeok Kim, SeongMin Park, GeumSook Heo, Joonhyouk Jang)

요 약

본 논문에서는 우주임무용 실시간 운영체제인 RTEMS QDP의 SMP 지원 현황과 주요 SMP 스케줄러의 동작 원리를 분석하였다. RTEMS 4.11.0 버전부터 SMP(Symmetric Multi-Processing)가 도입되었으나, 관련 연구는 미흡한 실정이다. 이에 본 연구에서는 RTEMS에서 지원하는 SMP 스케줄러들의 태스크 관리 및 할당, 로드 밸런싱 방식을 분석하고, 특히 EDF_SMP 스케줄러의 구현을 심층 분석하였다. 또한, 에뮬레이터 환경에서 단일 코어와 다중 코어 스케줄러를 비교한 결과, SMP 적용 시 긴 주기 태스크에서는 병렬 처리에 따른 성능 향상이 확인되었으나, 짧은 주기 태스크의 경우 스케줄러 오버헤드로 인해 성능 개선 폭이 제한적임을 확인하였다.

■ 중심어 : 스케줄러 ; 실시간 운영체제 ; 스케줄러 오버헤드

Abstract

In this paper, the current status of SMP support for RTEMS QDP, a real-time operating system for space missions, major SMP schedulers were analyzed. Symmetric Multi-Processing (SMP) was introduced from RTEMS 4.11.0 version, but related research is insufficient. Therefore, in this study, task management, allocation, and load balancing methods of SMP schedulers supported by RTEMS were analyzed, and in particular, the implementation of EDF_SMP scheduler was analyzed in depth. In addition, as a result of comparing single-core and multi-core schedulers in an emulator environment, it was confirmed that performance improvement was confirmed by parallel processing in long-period tasks when applying SMP, but in the case of short-period tasks, the performance improvement was limited due to scheduler overhead.

■ keywords : RTEMS QDP ; SMP ; schedulers

Ⅰ. 서 론

우주 임무용 컴퓨팅 시스템은 안전성 등의 이유로 단일 코어 코어를 기반으로 설계되어 왔으나, 최근에는 다중 코어 아키텍처가 본격적으로 채택되고 있다. 다중 코어 도입은 연산 성능과 에너지 효율 측면에서 분명한 이점을 제공하지만[11,12], 동시에 스케줄링 복잡성을 크게 증가시킨다[1]. 그러나, 우주 임무에서 요구되는 컴퓨팅 성능은 점차 비약적으로 증가하고 있다. 예를 들어, 고해상도 영상 처리·분석, 기계학습 기반 자율 운항, 실시간 과학 실험 제어, 분산 센서 융합 등 새로운 페이로드가 다수 탑재되면서 단일 코어 시스템으로는 처리량과 응답 시간을 만족시키기에는 한계가 있다. NASA의 차세대 고성능 우주용 컴퓨팅(HPSC) 문서에서는, 미래 미션의 컴퓨팅 수요를 충족하기 위해 강한 방사선에 노출되는 환경에서도 동작 가능한 멀티코어 HP

* 준회원, 한남대학교 전기·전자공학과
** 준회원, 한남대학교 컴퓨터공학과
*** 정회원, LIG 넥스원
**** 종신회원, 한남대학교 컴퓨터공학과
본 연구는 LIG Nex1의 "고신뢰성 실시간 멀티태스킹 지원 커널 개발 및 검증"(Y24-F005) 산학협력과제 지원으로 연구되었음..
접수일자 : 2025년 09월 02일 게재확정일 : 2025년 09월 17일
수정일자 : 2025년 09월 15일
교신저자 : 장준혁 e-mail : jhjang@hnu.kr

SC 코어를 반드시 채택해야 함을 제시하고 있다[5][10].

본 연구는 ECSS(유럽우주표준) 요구사항을 충족하기 위해 개발된 프레임워크인 RTEMS(Real-Time Executive for Multiprocessor Systems) QDP(Qualification Data Package) 환경에서 수행되었다. 우주 임무에 사용되는 RTEMS는 오픈 소스 실시간 운영체제로, RTEMS에서 지원하는 스케줄링 알고리즘은 SIMPLE SMP, EDF SMP, Priority SMP, Priority-Affinity SMP 등 다양한 SMP 스케줄러와 클러스터형 스케줄링을 통해 캐시 토폴로지에 최적화된 구성 옵션을 제공한다. 그러나, 현재까지 RTEMS에서 제공하는 다양한 SMP 스케줄링 알고리즘에 대한 연구는 제한적으로 이루어졌으며, 특히 실제 우주 비행체에 널리 사용되는 SPARC 계열 코어를 모사한 환경에서의 SMP 스케줄링 성능 분석 사례는 매우 부족한 실정이다. 이는 SMP 환경으로의 전환을 고려하는 우주 임무 설계자들에게 적절한 스케줄러 선택에 대한 명확한 기준과 가이드라인 제공을 어렵게 만드는 요인이 되고 있다.

본 연구는 RTEMS QDP에서 지원하는 SMP 스케줄러의 동작을 이해하고 그 특성을 파악하기 위해 수행되었다. 본 연구의 기여는 다음과 같다. 첫째, RTEMS의 각 SMP 스케줄러(SIMPLE_SMP, EDF_SMP, Priority_SMP, Priority_Affinity_SMP)의 동작 및 로드 밸런싱 메커니즘을 파악하였다. 둘째, 기본 SMP 스케줄러인 EDF_SMP 스케줄러의 코드 분석을 통해 SMP 환경에서 태스크 관리 및 할당 기법을 심층 분석하였다. 셋째, GR740 에뮬레이터인 sparc-rtems6-sis를 사용하여, 단일/다중 코어 환경의 스케줄링이 주기 태스크의 실행(처리량 및 응답 지연)에 미치는 영향을 비교하였다.

본 논문의 구성은 다음과 같다. 2장에서는 연구 배경에 대해 소개한다. 3장에서는 RTEMS SMP 스케줄러, 특히 EDF_SMP 스케줄러에 대해 상세히 다룬다. 4장에서는 동일 태스크로 UNI(단일 코어)/SMP(다중 코어) 스케줄링 특성을 분석한다. 5장에서는 결론과 향후 연구 방향에 대해 논의한다.

Ⅱ. 연구 배경

1. 단일 코어 실시간 스케줄링

단일 코어 환경에서, RM, EDF 스케줄러는 VxWorks나 RTEMS 등의 RTOS의 실시간 스케줄러로 널리 사용되고 있다. 특히 항공우주 분야와 같은 안전 필수 시스템에서는 ARINC 653 표준을 적용한 시간 분할(Time Partitioning) 사례가 있다. Orion 탐사선의 비행 소프트웨어(FSW)에서는 ARINC 653이 정의한 Major/Minor Time Frame 메커니즘을 활용하여 전체 CPU 시간을 고정된 Major Time Frame으로 구획하고, 각 파티션에 Minor Time Frame을 순차 할당함으로써 파티션 간 철저한 시간적 격리를 보장한다[6]. ARINC 653은 IMA 환경에서 시간·공간 분리(Time and Space Partitioning)를 보장하기 위해 정의된 국제 표준이다[2]. 이처럼 단일 코어 실시간 스케줄링 기법은 오랜 연구와 검증된 구현을 바탕으로 예측 가능성과 분석 용이성이라는 장점을 지니지만[7], 단일 코어의 처리 한계로 인해 우주 시스템의 복잡성과 연산 수요가 증가함에 따라 다중 코어 활용이 점차 부각되고 있다.

2. 다중 코어 실시간 스케줄링

다중 코어 환경에서는 스케줄링 복잡성이 크게 증가한다[3][9]. 실시간 태스크 스케줄링 방식은 글로벌 스케줄링과 클러스터드 스케줄링으로 구분된다. 글로벌 스케줄링은 모든 태스크를 하나의 공용 대기열에서 관리해 어떤 코어에서도 실행 가능하게 하는 방식이며, 클러스터드 스케줄링은 태스크를 각 코어에 고정 배정해 독립적으로 스케줄링한다. 글로벌 방식은 단일 코어에서 최적인 RM·EDF 알고리즘을 그대로 적용할 경우 특정 부하에서 이용률 저하라는 한계가 있고, 클러스터드 방식은 태스크의 최적 할당이 NP-h

ard 문제이다[4][8]. 절충안인 클러스터드 스케줄링은 일부 코어를 묶어 클러스터 내에서는 글로벌 스케줄링을, 클러스터 간에는 태스크 이동을 제한한다. 캐시 토폴로지를 반영해 코어를 그룹화하면 마이그레이션 횟수를 줄이고 캐시 결함 지연을 완화해, 글로벌 스케줄링 성능을 유지하면서 실시간 예측성을 높일 수 있다.

III. 본 론

1. RTEMS 태스크 모델

RTEMS에서의 태스크들은 (그림 1)와 같이 5가지 상태로 이루어진다. Non-existent 상태는 존재 하지 않는 상태로, 태스크 생성 전 또는 태스크 삭제 후에 태스크가 존재하지 않는 상태다. Dormant 상태는 태스크가 생성 되었지만, 태스크 시작이 호출 되지 않은 상태이다. Ready 상태는 스케줄링 대상이 되어 CPU점유를 요청할 수 있는 상태지만, 아직 선점되지 않은 상태이며, 동일 우선순위 태스크는 FIFO 방식으로 실행된다. Executing 상태는 현재 CPU를 점유하여 실행되고 있는 상태이다. Blocked 상태는 태스크가 작업을 수행할 수 없는 상태로, 예를 들어 메시지 큐, 세마포어 등에 의해 차단된 상태이다.

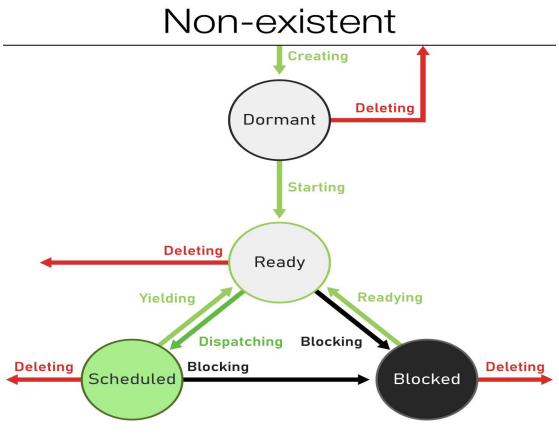

그림 1. RTEMS 태스크 상태

2. RTEMS SMP 스케줄링

가. RTEMS SMP 스케줄링 정책

RTEMS의 SMP 스케줄러는 모두 우선순위(priority) 기반으로 동작하며, 각 스케줄러 인스턴스는 자신이 관리하는 코어 집합에서 가장 높은 우선순위를 가진 태스크에 CPU를 할당한다. RTEMS SMP는 기본적으로 글로벌 스케줄링과 클러스터드 스케줄링 방식을 지원한다. 클러스터드 스케줄링은 시스템의 전체 코어를 둘 이상으로 분할하고, 각 코어 집합(클러스터)에 독립적인 스케줄러 인스턴스를 배치하는 방식이다. 이때 클러스터 크기가 1이면 파티션 스케줄링, 클러스터 크기가 시스템의 모든 CPU를 포함하면 글로벌 스케줄링이 된다.

이러한 구조는 멀티코어 환경에서 캐시 지역성을 향상시키고, 불필요한 태스크 마이그레이션을 감소시켜 실시간 성능을 개선하는 것을 목적으로 한다.

나. SMP 스케줄링 알고리즘

본 절에서는 RTEMS에서 지원하는 SMP 스케줄링 알고리즘 SIMPLE_SMP, EDF_SMP, Priority_SMP, Priority_Affinity_SMP 각각의 동작 및 특징에 대해 기술한다.

1) SIMPLE SMP

단순 우선순위 SMP 스케줄러로 하나의 정렬된 레디 큐를 사용하여 모든 준비 태스크를 우선순위 순으로 관리한다. 즉, 준비 상태의 태스크들이 단일 연결리스트에 우선순위에 따라 정렬되어 있으며, (만약 시스템에 N개의 코어가 있다면) 실행 가능 태스크 중 최상위 우선순위 태스크들이 상위 N개까지 CPU에 할당된다. 새로운 태스크가 준비 상태로 들어오거나 우선순위 변화가 발생하면 해당 리스트에 삽입되며, 이 때 연결리스트 삽입 연산이 준비 태스크 수에 비례하는 O(n)의 시간복잡도를 갖는다. SIMPLE_SMP 스케줄러는 구현이 단순한 대신, 우선순위별 별도 자료구조를 사용하지 않아 메모리 사용은 적지만 결정론적 응답시간 보장은 어렵다.

2) EDF SMP

EDF_SMP 스케줄러는 동적 우선순위 스케줄러에 속한다. 각 태스크는 데드라인을 우선순위 결정에 사용하며, 활성화된 데드라인이 있는 태스크들은 그렇지 않은 태스크보다 더 높은 우선순위를 갖는다. 데드라인이 설정되지 않은 태스크들은 백그라운드 태스크로 간주되어 고정 우선순위 방식(정적 우선순위)에 따라 스케줄링되며, 시스템에서 데드라인을 전혀 사용하지 않으면 일반적인 우선순위 스케줄러처럼 동작한다. EDF_SMP에서는 태스크 Ready 큐로 레드-블랙 트리를 사용하며, 이 트리를 태스크의 현재 우선순위에 따라 정렬된다. 따라서 스케줄링 연산이 O(log n)의 복잡도로 비교적 효율적이다.

3) Priority SMP

Priority_SMP 스케줄러는 결정론적 우선순위 SMP 스케줄러로, 글로벌 고정 우선순위 스케줄러에 해당한다. 이 스케줄러는 우선순위 수준별로 별도의 큐를 유지하는 방식으로 구현되어 있다. 시스템의 최대 우선순위 개수(기본 0~255)만큼 준비 리스트가 존재하며 각 리스트는 해당 우선순위의 준비 태스크들을 FIFO 순서로 보유한다. 스케줄링 시에는 가장 높은 우선순위 리스트부터 확인하여 실행 가능 태스크를 선택하며, 최상위 우선순위에 여러 태스크가 대기하면 그 중 상위 N개 태스크가 N개의 CPU에 할당 된다. 이때 우선순위 비트맵을 이용하여 가장 높은 우선순위 레벨을 빠르게 찾아내므로, 스케줄링 결정에 소요되는 시간이 태스크 수에 관계없이 우선순위 수준의 개수에만 비례하는 O(1)의 상한을 갖는다.

4) Priority Affinity SMP

Priority_Affinity_SMP 스케줄러는 임의 코어 친화도 우선순위 SMP 스케줄러로 Priority_SMP 스케줄러에 태스크별 코어 친화도 설정을 추가로 지원하는 스케줄러이다. 기본적인 구조는 Priority_SMP와 유사하게 우선순위별 준비 리스트를 사용한 고정 우선순위 스케줄링이며, 최대 우선순위 및 데이터 구조도 같다. 다만 각 태스크마다 실행 가능 코어의 부분집합을 임의로 지정할 수 있고, 스케줄러는 해당 태스크가 실행될 수 있는 CPU에서만 스케줄링되도록 할당을 결정한다. 이러한 유연성 때문에 스케줄링 알고리즘의 복잡도가 증가하는데, 특정 CPU에 대해 실행할 수 없는 태스크들을 건너뛰고 다른 태스크를 찾아야 하는 등 추가 연산이 필요하므로 일부 스케줄링 연산의 시간 복잡도는 최악의 경우 O(n)까지 늘어날 수 있다.

다. SMP 스케줄링 의사 결정

리눅스 등의 운영체제는 로드 밸런싱과 마이그레이션을 위해 주기적으로 실행되는 스레드가 별도로 존재한다. 반면 RTEMS는 스케줄러가 태스크 상태 변화 시점에 즉각적으로 결정을 내림으로써 다중 코어 간의 작업 분배를 효율적으로 수행한다.

1) 로드 밸런싱

글로벌 스케줄링의 경우, RTEMS 스케줄러는 항상 가능한 모든 코어를 바쁘게 유지하도록 동작한다. 한 코어가 유휴 상태가 되면 즉시 현재 준비 큐에 대기중인 가장 높은 우선순위 태스크를 해당 코어에 할당하고 실행시킨다. 반대로 여러 태스크가 동시에 준비 상태가 되면, 그 중 최고 우선순위를 태스크들부터 사용 가능한 CPU에 분배되어 병렬 실행된다.

클러스터드 스케줄링에서는 로드 밸런싱이 클러스터 내부에서만 이루어지며, 태스크와 코어가 각각 소속된 클러스터를 넘지 않는다. 예를 들어 한 클러스터에 속한 코어들이 모두 유휴 상태여도, 다른 클러스터에 속한 태스크를 가져와 실행하지는 않는다.

2) 마이그레이션

마이그레이션이란 태스크의 실행 코어가 변경되는 현상을 뜻하며, RTEMS에서는 다음과 같은 네 가지 경우에 발생할 수 있다.

첫째로 명시적 스케줄러 변경이다. 애플리케이션이 rtems_task_set_scheduler() 등 API 호출을 통해 태스크를 다른 스케줄러 인스턴스로 이동시킬 경우, 해당 태스크는 새로운 CPU로 즉시 마이그레이션 된다.

둘째로는 명시적 코어 친화도 변경이다. rtems_task_set_affinity() 등을 API 호출을 통해 태스크의 허용 코어 집합을 변경하면, 그 태스크가 현재 속한 CPU가 새로운 친화도에 포함되지 않는 경우 스케줄러는 태스크를 해당 범위 내의 다른 CPU로 이동시킨다. 즉, 사용자가 강제로 친화도를 조정하면 태스크가 마이그레이션 될 수 있다.

셋째로는 태스크 unblock이다. 태스크가 이벤트 대기나 자원 잠금등으로 blocked 상태였다가 깨워질 때, 해당 태스크는 다시 Ready가 된다. 이 때 깨어난 태스크의 우선순위가 현재 실행 중인 어떤 태스크보다 높다면, 즉시 가장 낮은 우선순위의 실행 태스크를 밀어내고 깨어난 태스크를 대신 실행시킨다.

마지막으로 특수한 lock 프로토콜이다. RTEMS SMP는 클러스터 간 자원 공유 시 MrsP(Multiprocessor Resource Sharing Protocol)이나 OMIP(O(m) Independence-Preserving Protocol) 같은 분산 locking 프로토콜을 사용해 우선순위 역전을 방지한다. 락을 가진 태스크는 필요한 클러스터로 일시 마이그레이션해 실행 후 복귀하며, 그 외에는 캐시 효율을 위해 초기 코어에서 계속 실행한다.

라. EDF_SMP 스케줄러 분석

본 절에선 EDF_SMP 스케줄러가 RTEMS 내에서 어떻게 구현되었는지 확인하기 위해 코드 분석을 수행하였다.

1) EDF_SMP 코드 분석

RTEMS rtems-6-sparc-gr712rc-smp6 내부에는 EDF_SMP를 구현한 'scheduleredfsmp.h' 헤더 파일이 존재하며, Scheduler_EDF_SMP_Node, Scheduler_EDF_SMP_Ready_queue, Scheduler_EDF_SMP_Context 세 개의 구조체가 정의되어 있다. 본 절에서는 위 세 가지 구조체를 중심으로 코드 구현 분석과 실제 EDF_SMP의 실행 흐름을 함께 분석한다. 이후 세 구조체를 각각 EDF_SMP_Node, EDF_SMP_Ready_queue, EDF_SMP_Context로 약칭한다.

A. Scheduler_EDF_SMP_Node

Node는 태스크의 스케줄링 표현체로, 레디 큐에 삽입되어 CPU 할당 여부를 판단받는 구조체 단위이다. EDF_SMP_Node는 SMP용 스케줄러 노드인 Scheduler_SMP_Node를 포함한다. Scheduler_SMP_Node는 현 노드의 우선순위값과 BLOCKED/SCHEDULED/READY의 노드의 상태 필드를 가지며, 공통 스케줄러 인터페이스인 Scheduler_Node를 포함하는 구조체이다. EDF_SMP_Node 구조체는 Scheduler_SMP_Node 포함과 함께 추가 필드로 generation, ready_queue_index, affinity_ready_queue_index, pinning_ready_queue_index를 가지고 있다.

generation은 노드 간 비교를 위한 필드로, Chain_Control Scheduled에서 노드를 정렬할 때 비교 요소로 사용된다. Chain_Control Schedule은 SCHEDULED 상태의 노드들을 담는 이중연결 리스트이다. Chain_Control Scheduled에선 노드 비교 시 (그림 2)와 같이 먼저 priority 값을 비교하며, priority 값이 같다면 generation 값이 더 작은 노드가 체인의 앞쪽에 배치된다.

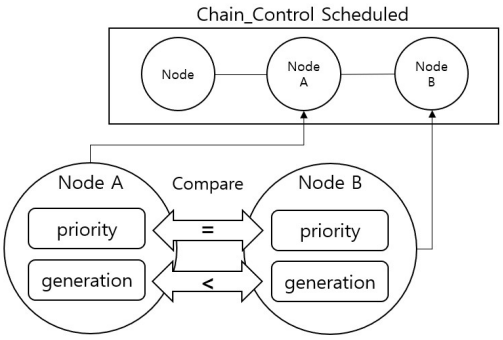

그림 2. generation 비교

ready_queue_index는 노드가 현재 실제로 들어가 있는 레디 큐의 인덱스를 저장하는 필드이다. EDF_SMP는 멀티코어 환경에 따라 여러 개의 레디 큐가 존재하며, 각 레디 큐는 레디 큐 배열 안에서 관리된다. 0번 인덱스는 공용 큐이며 1~N번 인덱스는 CPU 전용 큐이다. ready_queue_index의 값이 0일 경우 노드는 공용 큐에 속하며, 1~N이면 특정 CPU 큐에 속해 있음을 나타낸다. 공용 큐는 모든 CPU가 공유하는 레디 큐이며, CPU 전용 큐는 특정 CPU만 허용하는 레디 큐이다.

affinity_ready_queue_index는 노드가 특정 CPU 큐에 고정되지 않았을 경우, 해당 노드의 선호 레디 큐 인덱스를 저장하는 필드이다. CPU에 고정된 상태가 아니라면 affinity_ready_queue_index 값은 즉시 ready_queue_index에도 반영된다. 고정된 상태라면 선호 큐 인덱스값은 affinity_ready_queue_index에 저장만 해두며, 특정 CPU 큐에서 고정 해제 시 affinity_ready_queue_index 필드에 저장해 둔 값을 ready_queue_index로 반영한다.

pinning_ready_queue_index는 노드가 고정된 특정 CPU 큐의 인덱스를 저장하는 필드이다. CPU 큐에 노드가 고정되면 pinning_ready_queue_index와 ready_queue_index가 모두 해당 CPU 큐의 인덱스값으로 설정된다. 노드가 CPU 큐에서 고정 해제 시 pinning_ready_queue_index 값은 0이 되며 ready_queue_index에는 affinity_ready_queue_index 값을 반영한다.

B. Scheduler_EDF_SMP_Ready_queue

EDF_SMP_Ready_queue는 레드-블랙 트리 형태의 노드를 저장하는 레디 큐를 포함한 구조체이다. EDF_SMP_Ready_queue 구조체를 구성하는 필드는 Chain_Node, RBTree_Control, Scheduler_EDF_SMP_Node *affine_scheduled, Scheduler_EDF_SMP_Node *allocated가 있다.

Chain_Node는 CPU 전용 큐를 Affine_queues에 연결 및 해제하는 링크 역할의 필드이다. Affine_queues는 CPU 전용 큐가 전역 최상 우선순위 노드 선정 과정에 참여할 수 있도록 해주는 체인이며, 전역 최상 우선순위 노드 선정 과정은 공용 큐의 가장 높은 우선순위의 노드를, Affine_queues 체인에 연결된 모든 CPU 전용 큐를 순회하며 각 큐의 가장 높은 우선순위 노드와 비교하는 과정이다.

RBTree_Control은 노드가 삽입되는 실제 레디 큐 구조체이며, 레드-블랙 트리 구조를 가진다. 노드는 RBTree_Control에 데드라인 우선순위 순으로 정렬되어 들어가며 우선순위가 높을수록 트리의 좌측에 배치된다. 전역 최상 우선순위 노드 선정 과정에선 (그림 3)과 같이 공용 큐의 RBTree_Control 내 좌측 끝 노드와, CPU 전용 큐의 RBTree_Control 내 좌측 끝 노드를 비교한다. 공용 큐 노드와 CPU 전용 큐 노드 비교 시 priority 값과 generation 값이 활용된다. 레디 큐에서 노드를 제거하거나 재삽입을 하는 과정도 RBTree_Control을 대상으로 한다.

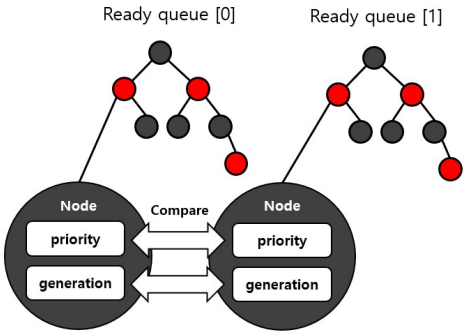

그림 3. 좌측 끝 노드 비교

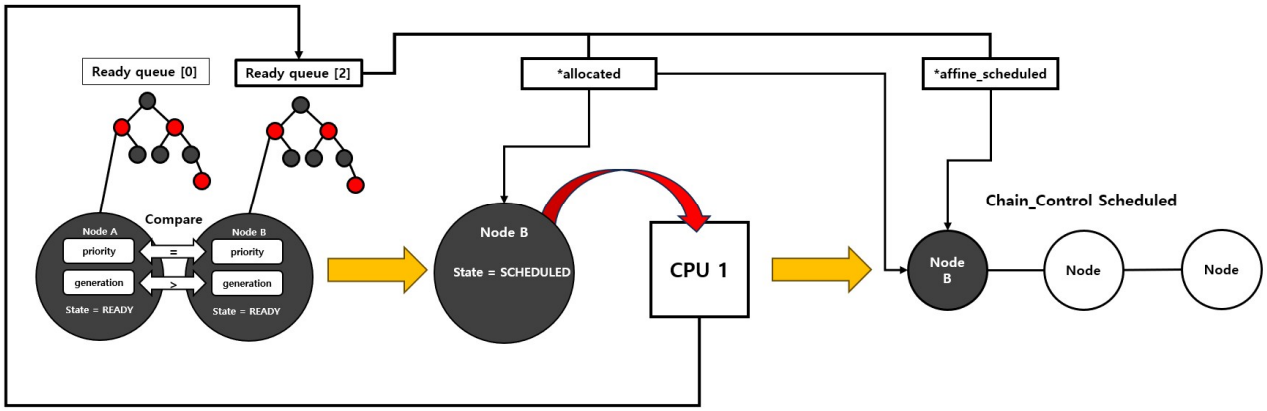

그림 4. *affine_scheduled 포인터와 *allocated 포인터의 노드

Scheduler_EDF_SMP_Node *affine_scheduled는 CPU 전용 큐에서 SCHEDULED 상태인 노드를 가리키는 포인터 필드이다. CPU 전용 큐의 가장 높은 우선순위 노드가 전역 최상 우선순위 노드로 선정되면, 해당 노드는 READY에서 SCHEDULED로 상태 전환 후 실제 CPU에 배치되며 Chain_Control Scheduled 체인에 삽입된다. 체인 삽입 후 (그림 4)와 같이 해당 CPU 전용 큐의 Scheduler_EDF_SMP_Node *affine_scheduled 필드가 해당 노드를 가리킨다.

Scheduler_EDF_SMP_Node *allocated는 CPU에 현재 배치될 SCHEDULED 상태의 노드를 가리키는 포인터 필드이다. 노드의 상태가 READY에서 SCHEDULED로 전환되면, (그림 4)와 같이 배치될 CPU의 전용 레디 큐 내의 Scheduler_EDF_SMP_Node *allocated 필드가 해당 노드를 가리킨 후 해당 노드를 실제 CPU에 배치한다. 이후 해당 노드를 Chain_Control Scheduled 체인에 삽입한다.

C. Scheduler_EDF_SMP_Context

EDF_SMP_Context는 공용 큐와 CPU 전용 큐를 담는 레디 큐 배열과, CPU 전용 큐들을 전역 최상 우선순위 노드 선정 과정에 참여시키는 체인을 포함하는 구조체이다. EDF_SMP_Context를 구성하는 필드는 Scheduler_SMP_Context Base, generations[2], Chain_Control Affine_queues, Scheduler_EDF_SMP_Ready_queue Ready[RTEMS_ZERO_LENGTH_ARRAY]가 있다.

Scheduler_SMP_Context Base는 SMP 공통 스케줄러를 지원하는 구조체로, 해당 구조체는 공유자원 접근을 방지하기 위한 Lock과 CPU 집합을 관리하는 Processor_mask를 포함하는 Scheduler_Context 구조체를 가지며, SCHEDULED 상태의 노드들을 담는 Chain_Control Scheduled 구조체를 포함한다.

generations[2]는 노드의 generation 값을 갱신하기 위한 배열 필드이며, 증가와 감소로 동작한다. 해당 필드는 Chain_Control Scheduled 체인에서 노드를 정렬할 때와, 전역 최상 우선순위 노드 선정 과정의 노드 비교에서 사용된다. gene

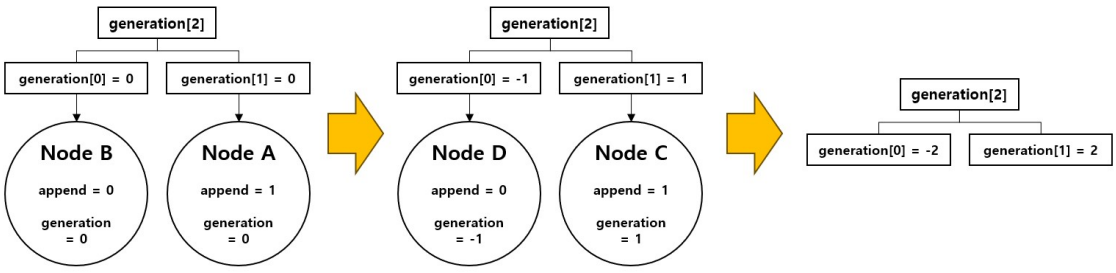

그림 5. generation 갱신 과정

rations[2]의 0번 인덱스는 값이 1씩 감소되며, 1번 인덱스는 값이 1씩 증가한다. generations[2] 필드는 Chain_Control Scheduled 체인에서 노드 정렬 직전에 generation 값을 노드에 부여하며, 인덱스 0과 1을 노드별로 분류하여 generation 값을 갱신시킨다. 분류 기준은 노드의 append 플래그의 차이이다. append 플래그는 노드의 우선순위를 나타내는 정수형 필드인 Priority_Control 값의 하위 1비트를 이용하며, 해당 비트가 1이라면 generations의 1번 인덱스 그룹, 0이면 0번 인덱스 그룹으로 분류한다. 노드의 generation 갱신 예시는 (그림 5)와 같다. generations의 0, 1 인덱스가 모두 0으로 초기화된 상태에서 노드 A의 append가 1일 경우, generations[1]이 노드 A의 generation 값을 0으로 갱신 후 generations[1]은 값 1로 갱신된다. 노드 B 의 append가 0일 경우 generations[0]은 노드 B의 generation 값을 0으로 갱신 후 generations[0] 값은 −1이 되며, 노드 C 의 append가 1이면 generations[1]은 노드 C의 generation 값을 노드 A 때 갱신한 1로 부여한 뒤 generations[1] 값은 2가 된다. 노드 D의 append가 0이면 generation[0]은 노드 D의 generation 값을 노드 B 때 갱신한 −1로 부여하고 generations[0] 값은 −2로 갱신된다.

Chain_Control Affine_queues는 CPU 전용 큐가 전역 최상 우선순위 노드 선정 과정에 참여할 수 있도록 해주는 체인 역할의 필드이다. 해당 체인에 등록되는 조건은 (그림 6)과 같다. SCHEDULED 상태인 노드와 READY 상태인 노드가 없는 CPU 전용 큐에, 새로운 노드가 들어올 경우 해당 체인에 등록된다. SCHEDULED 상태의 노드와 READY 상태의 노드를 가지고 있는 CPU 전용 큐는 일시적으로 해당 체인에서 제외되며, SCHEDULED 상태의 노드와 READY 상태의 노드가 둘 다 없는 CPU 전용 큐도 일시적으로 Affine_queues에서 제외된다. 그러나 SCHEDULED 상태의 노드와 READY 상태의 노드가 있는 CPU 전용 큐에서, SCHEDULED 상태의 노드가 READY 상태로 전환될 경우 해당 큐는 해당 체인에 재등록된다.

Scheduler_EDF_SMP_Ready_queue Ready[RTEMS_ZERO_LENGTH_ARRAY]는 공용 레디 큐와 CPU 전용 레디 큐를 담는 배열 필드이다. 해당 필드의 배열 내엔 RTEMS_ZERO_LENGTH_ARRAY 매크로가 선언되어 있어, 실제 메모리 할당 시점에 필요한 개수만큼 레디 큐 추가가 가능하다. 해당 배열 필드는 0번 인덱스에 공용 큐를 배치하며, CPU 전용 큐는 1~N번 인덱스에 배치한다.

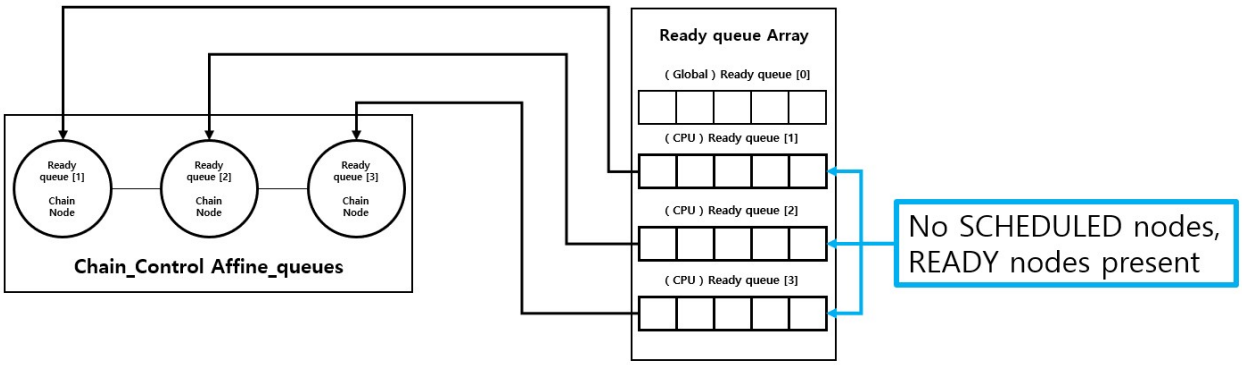

그림 6. Chain_Control Affine_queues 등록

Ⅳ. 실험

1. 실험 환경 및 태스크 구성

코어 수가 늘어났을 때, 병렬화 효과를 확인하기 위해 단일 코어(UNI)와 다중 코어(SMP) 환경에서 동일 스케줄러(EDF_SMP) 하의 처리량, 응답 지연, CPU 점유 특성을 실험하여 비교 분석했다.

가. 실험 환경

실험은 Ubuntu 22.04.5 LTS 환경에서 수행되었으며, VS Code의 Remote-SSH를 이용해 테스트 코드를 편집·디버깅하였으며, 빌드는 RTEMS QDP GR740 SMP V6, UNI V6를 대상으로 waf 스크립트로 수행하였다. 실행 파일 구동은 Gaisler Research의 sparc-rtems6-sis 시뮬레이터에서 UNI, SMP(4-Core) 모드로 로드되어 실행되었다. rtems_cpu_usage_report와 rtems_rate_monotonic_report API를 통해 태스크별 CPU 점유율, CPU Time, Wall Time을 측정하였다. 이로써 단일 코어와 다중 코어 모드에서 동일한 스케줄러 하에 발생하는 성능 차이를 하드웨어 의존성 없이 객관적으로 비교·분석하였다.

나. 태스크 구성

실험에 사용한 워크로드는 실제 우주 임무 사례를 참조하여 모두 주기적 태스크로 구성하였다. (표 1)과 같이 총 7개의 태스크를 정의하였으며, 각 태스크는 임무 기능에 따라 서로 다른 주기와 고정된 실행 시간을 가진다. T1, T2는 CCDSS-123 표준 기반의 온보드 영상 압축 적용 사례를 참조하여 500ms 주기를 부여하였다[16]. T3, T4는 EIVE 큐브셋의 ADCS(자세제어 시스템) 운용 주기를 참조하여 100ms, 400ms로 설정하였다[17]. T5는 센서 데이터 처리율과 동일하게 100ms로 설정하였다. T6, T7은 NASA cFS Scheduler 문서에서 제시하는 구조를 참조하여 1000ms 주기를 부여하였다[18].

실행 시간은 문헌에서 직접적인 수치를 제시하지 않으므로, 단일 코어 기준 CPU 사용률이 약 84%가 되도록 설정하였다.

표 1. 태스크 구성

태스크	주기 (ms)	실행 시간 (ms)	기능
T1	500	60	영상 압축
T2	500	60	영상 전처리 및 패킹
T3	100	12	현재 자세 추정
T4	400	48	자세 제어 명령
T5	100	12	통신
T6	1000	120	상태 텔레메트리
T7	1000	120	로그 기록 및 파일 I/O

다. 측정 항목

UNI, SMP 환경 간의 처리량 차이를 비교하고, 스케줄러 오버헤드와 응답 지연을 측정하기 위해 측정 항목은 각 태스크 당 실행 횟수, 평균 CPU Time, 평균 Wall Time으로 정의하였다. CPU Time은 태스크가 실제로 CPU를 점유하여 연산된 시간을 집계하고, Wall Time은 컨텍스트 스위칭, 인터럽트 처리 등 오버헤드들과 같은 지연도 포함되어 측정된다. 위 측정 항목들은 rtems_rate_monotonic_report_statistics_withj_plugin() API를 통해 수집하였다. 위 API는 rtems_rate_monotonic_period() 호출을 통해 동작한다.

2. 실험 결과

4.1 절에서 정의한 태스크 구성을 기반으로 UNI, SMP(4-Core)환경에서 각각 실행한 후 수집한 태스크 실행 횟수, Wall Time, CPU Time 결과를 분석한다.

가. 실행 횟수

태스크별 실행 횟수 측정 결과(그림 7)을 살펴보면, 모두 주기 기반 태스크이므로 대부분 동일한 실행 횟수를 보였다. 그러나 SMP 전환 시 짧은 주기의 T3은 2회, T5는 3회 더 많은 실행을 기록하였다. 이는 단일 코어 환경에서 일부 주기 작업이 기한 내 실행 되지 못한 반면, 다중 코어 환경에서는 태스크가 여러 코어에 병렬 분산된 모습을 나타낸다.

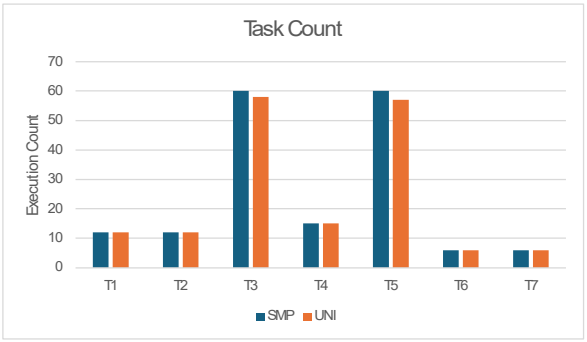

그림 7. 태스크 별 실행 횟수

나. 평균 Wall Time

태스크별 평균 Wall Time 측정 결과(그림 8)은 다중 코어 환경이 단일 코어 환경보다 전체적으로 Wall Time이 짧거나 유사한 경향을 보였다. 긴 주기를 가진 T1, T2는 Wall Time이 22% ~ 28% 감소하여 응답 지연이 개선되었고, T4, T7도 각각 3% ~ 4% 감소하였다. 반면, 짧은 주기를 가진 T3, T5, T6은 Wall Time 감소 폭이 0.5% 이내로 미미하다. 그 이유는 짧은 주기의 태스크일수록 스케줄러 오버헤드가 전체 실행 시간에서 차지하는 비율이 늘어나고, 실제로 유의미한 연산을 수행하는 시간 대비 스케줄러가 일하는 시간이 더 커지기 때문이다.

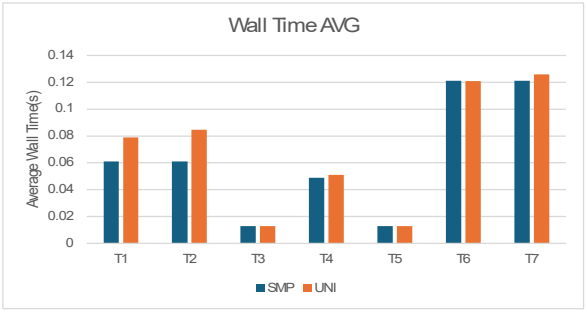

그림 8. 태스크별 평균 Wall Time

다. 평균 CPU Time

태스크별 평균 CPU Time 측정 결과(그림 9)는 다중 코어 환경에서의 측정값이 단일 코어 환경에서의 측정값보다 뚜렷히 높게 측정되었다. CPU Time은 태스크가 실제 CPU 위에서 연산한 시간이므로 단일 코어 환경에서는 다른 태스크들과 경쟁으로 인해 예정된 WCET가 온전히 보장받지 못하였고, 다중 코어 환경에서는 각 태스크가 설계된 WCET만큼 안정적으로 CPU를 점유했음을 보였다.

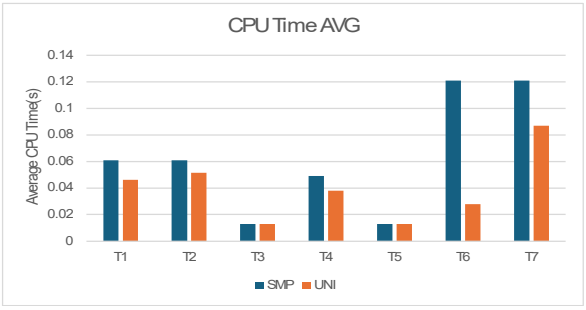

그림 9. 태스크별 평균 CPU Time

3. 고찰

본 연구를 통해 sparc-rtems6-sis 에뮬레이터 기반 RTEMS QDP GR740 환경에서 UNI/SMP 비교를 수행한 결과 다음과 같은 결과를 얻었다.

UNI에서 SMP로 전환 시 긴 주기 태스크(T1 - T4)는 병렬 분산 실행을 통해 실행 횟수와 CPU Time이 유의미하게 증가하며 처리량이 향상된 반면, 짧은 주기 태스크(T5 - T7)는 스케줄러 오버헤드가 상대적으로 커져 성능 병목이 발생함을 확인하였다. 이는 다중 코어 도입 전 태스크 실행 시간 대비 스케줄링 오버헤드 비율을 사전 분석해야 함을 알 수 있었다.

V. 결론

우주 임무에서 고해상도 영상 처리, 기계학습 기반 자율 운항, 실시간 과학 실험 제어 등 연산 수요가 증가함에 따라 SMP 도입은 필수적이다. 본 연구에서는 RTEMS QDP GR740 SMP 환경에서 SMP 핵심 구현 코드와 스케줄러 구조를 분석하고, sparc-rtems6-sis 에뮬레이터를 활용하여 동일 워크로드 하에서 UNI와 SMP 환경의 성능을 비교하였다. 실험 결과, SMP 전환 시 긴 주기 태스크의 처리량과 응답 성능이 크게 향상되었으나, 짧은 주기 태스크에서는 스케줄러 오버헤드로 인해 Wall Time 개선

폭이 제한적이었다. 이를 통해 SMP 적용 시 태스크 특성에 따른 성능 편차와 오버헤드 영향을 고려한 설계의 필요성을 확인하였다. 향후 연구에서는 클러스터드 스케줄링이나 혼합 정책을 적용해 다양한 워크로드 특성에 최적화된 스케줄러 구성을 검토할 예정이다.

REFERENCES

[1] A. VanderLeest and T. Thompson, “Measuring the Impact of Interference Channels on Multicore Avionics,” arXiv preprint arXiv:2104.01234, 2021.

[2] P. Han, Z. Zhai, B. Nielsen, and U. Nyman, “Model-based optimization of ARINC-653 partition scheduling,” *Softw. Tools Technol. Transf.,* vol. 23, no. 5, pp. 721 - 740, Oct. 2021.

[3] J. Carpenter et al., “A Categorization of Real-time Multiprocessor Scheduling Problems and Algorithms,” *Tech. Rep. TR07-001,* Univ. of North Carolina, Chapel Hill, 2007.

[4] A. C. Lindsay, “A Cache-Aware Multi-core Real-Time Scheduling Algorithm,” *M.S. thesis, Virginia Tech, Blacksburg,* VA, 2012.

[5] NASA, “High Performance Spaceflight Computing (HPSC) Concept and Technology Reference Mission,” *NASA/TM-2010-216164,* Jun. 2010.

[6] A. Lagoy and T. A. Gauer, “IV&V on Orion’s ARINC 653 Flight Software Architecture,” *NASA IV&V Facility Technical Report NASA/TP - 2010 - 216123,* Jul. 2010.

[7] C. L. Liu and J. W. Layland, “Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment,” *J. ACM,* vol. 20, no. 1, pp. 46 - 61, Jan. 1973.

[8] S. K. Baruah, N. K. Cohen, C. G. Plaxton, and D. A. Varvel, “Proportionate Progress: A Notion of Fairness in Resource Allocation,” *Algorithmica,* vol. 15, no. 6, pp. 600 - 625, Jun. 1996.

[9] R. Davis and A. Burns, “A Survey of Hard-Real-Time Scheduling for Multiprocessor Systems,” *ACM Comput. Surv.,* vol. 43, no. 4, Art. 35, pp. 35:1 - 35:44, Oct. 2011.

[10] A. S. Keys et al., “Developments in Radiation-Hardened Electronics Applicable to the Vision for Space Exploration,” *NASA Marshall Space Flight Center Technical Report,* 2007.

[11] G. Hager, J. Treibig, J. Habich, and G. Wellein, "Exploring performance and power properties of modern multicore chips via simple machine models," *Concurrency: Pract. Exper.,* vol. 28, no. 2, pp. 189 - 210, 2016.

[12] A. Butko, F. Bruguier, D. Novo, A. Gamatié, and G. Sassatelli, “Exploration of Performance and Energy Trade-offs for Heterogeneous Multicore Architectures,” arXiv preprint arXiv:1902.02343, 2019.

[13] A. Gujarati, F. Cerqueira, and B. B. Brandenburg, “Multiprocessor real-time scheduling with arbitrary processor affinities,” *in Proc. IEEE Real-Time Systems Symp. (RTSS),* pp. 247 - 258, Dec. 2013.

[14] G. Talmale and U. Shrawankar, “Comparative Analysis of Different Techniques of Real-Time Scheduling for Multi-Core Platform,” arXiv preprint arXiv:2112.13841, 2021.

[15] B. Sun, D. Roy, T. Kloda, and A. Bastoni, "Co-Optimizing Cache Partitioning and Multi-Core Task Scheduling: Exploit Cache Sensitivity or Not?," *in Proc. *2023 IEEE Real-Time Systems Symposium (RTSS)*,* 2023.

[16] CCSDS, “Low-Complexity Lossless & Near-Lossless Multispectral & Hyperspectral Image Compression (CCSDS 123.0-B-2),” *Blue Book,* Feb. 2019.

[17] M. T. Koller, L. Bötsch-Zavřel, C. Holeczek et al., “Next on the Pad: The E-band Technology Demonstration CubeSat EIVE,” *in 73rd International Astronautical Congress (IAC),* Paris, France, Sep. 2022.

[18] NASA GSFC, Core Flight System (cFS) Scheduler User’s Guide, *Technical Report,* 2020.

저 자 소 개

박주찬(준회원)

2020년~현재 한남대학교 전기전자공학과 학사 재학.
2025년~현재 한남대학교 우주공학과 학석사 연계과정.

<주관심분야 : 운영체제, 시스템소프트웨어, 인공위성>

김권혁(준회원)

2024년 한국폴리텍대학 AI융합소프트웨어과 산업학사 졸업.
2024년~2025년 ㈜포커스블루 머신비전 엔지니어.
2025년~현재 한남대학교 컴퓨터공학과 학사과정.

<주관심분야 : 운영체제, 시스템소프트웨어, 인공지능>

박성민(정회원)

2009년 홍익대학교 전자전기공학부 공학석사 졸업.
2011년 홍익대학교 전자정보통신공학과 공학석사 졸업.
2011년~현재 LIG 넥스원(주) 정지궤도위성 개발단 3팀 수석 연구원

<주관심분야 : 임베디드 시스템, 운영체제, 시스템 소프트웨어, 위성 비행 소프트웨어, 위성 시스템, 전파산란>

허금숙(정회원)

1996년 한림대학교 전자계산학과 이학사 졸업.
1996년~2007년 정보통신 분야 임베디드 SW 개발.
2011년~2019년 LIG 시스템(주) 무기체계분야 임베디드 SW 개발.
2020년~현재 LIG넥스원(주) 정지궤도위성 개발단 3팀 수석 연구원

<주관심분야 : 임베디드 SW, 운영체제, 시스템 소프트웨어, 위성 비행 소프트웨어, 위성 시스템>

장준혁(종신회원)

2009년 서울대학교 전기·컴퓨터공학부 학사 졸업.
2015년 서울대학교 전기·컴퓨터공학 박사 졸업(석박사통합).
2021년~현재 한남대학교 컴퓨터공학과 교수

<주관심분야 : 임베디드시스템, 운영체제, 시스템 소프트웨어>