논문 2014-09-31

인공위성 탑재컴퓨터를 위한 리눅스 기반 ARINC 653 공간 분리

(Linux-based ARINC 653 Space Separation for Spacecraft Computer)

김 덕 수, 조 현 우, 김 형 신*

(Duksoo Kim, Hyunwoo Joe, Hyungshin Kim)

Abstract : European Space Agency has recognized Integrated Modular Avionics and ARINC specification 653 as avionics computer system for space application. Integrated Modular Avionics specification reduces the space by integrating a system composed of many electronic devices into a computer. recent researches have been studying how to apply the ARINC 653 into an open source operating system, such as Linux. These studies have concentrated on partition scheduling for time separation. However, requirements to guarantee spatial separation should be further analyzed to ensure deterministic execution time. Therefore, memory management is needed to verify spatial isolation on Linux systems. This research proposes a new method to accomplish spatial isolation for the ARINC 653 specification in Linux. We have added new data structures and system calls to handle functionalities for spatial separation. They are used during the partition *startup process*. The proposed method was evaluated on the LEON4 processor, which is the next generation microprocessor to be used in the future space missions. All implementations confirm that spatial isolation of the ARINC 653 specification was accomplished

Keywords : Integrated Modular Avionics, ARINC 653, Linux, Space Separation, Memory

1. 서론

통합모듈항공전자 시스템(Integrated Modular Avionics)은 여러 전자 모듈로 구성된 시스템을 단일 컴퓨터로 통합하여 모듈이 차지하는 공간을 줄일 수 있다[1]. 우주용 전자 시스템에도 같은 이유로 통합모듈항공전자 시스템이 적용되고 있다[2-4]. 통합모듈항공전자 시스템은 독립된 인증 수준을 가진 모듈들이 같은 컴퓨터 자원을 공유한다. 이에 따라 시스템 검증 비용이 상승하는 문제가 발

생한다[5, 6]. 위와 같은 문제를 해결하기 위해서 파티셔닝 기술이 제안되었다. 파티션은 각 모듈의 실행과 고장이 서로 영향을 주지 않기 위해 하드웨어 자원을 시간/공간적으로 분리해 할당한다[2, 3]. 가장 잘 알려진 통합모듈항공전자 시스템의 구조로는 ARINC 653표준이 있다[6].

현재 ARINC 653은 VxWorks 653[7], Xtratum[8], AIR[9]와 같은 운영체제에 적용되고 있다. VxWorks 653과 같은 상용 운영체제는 높은 사용료와 유지보수 비용이 필요하다는 단점이 있으며, Xtratum, AIR와 같은 운영체제 또한 다양한 하드웨어에서 사용할 수 없다는 단점이 있다. 이에 따라 오픈 소스 리눅스에 ARINC 653표준을 적용하려는 연구가 소개되고 있다[10, 11]. Xtratum의 경우 MMU(Memory Management Unit)를 이용한 공간 분리를 제공하고 있으며, AIR의 경우에는 공간 분리를 지원하지 않는다. 리눅스에 ARINC 653표준을 적용하는 연구들은 시간분리를 위한 파티션 스케줄링에 집중되어 있으며, 공간분리에 관한 연구

*Corresponding Author(hyungshin@cnu.ac.kr)

Received: 1 Apr. 2014, Revised: 29 Apr. 2014, 3 June 2014, Accepted: 23 June 2014.

D.S. Kim, H.W. Joe, H.S. Kim: Chungnam National University

※본 연구는 한국연구재단을 통해 교육과학기술부의 우주기초원천기술개발 사업(NSL, National Space Lab)으로부터 지원받아 수행되었습니다. (2011-0020905).

결과는 미비하다. 때문에 리눅스의 메모리 관리 방법이 ARINC 653표준의 공간분리를 만족하는지에 대한 검증이 추가적으로 필요하다[11].

ARINC 653표준은 공간분리를 보장을 위해서 각 파티션에 할당된 메모리 영역을 서로 침범하지 않아야 하며, 할당된 메모리 자원이 운영체제에 의해 회수되지 않고 유지되어야 한다고 규정하고 있다[12, 13]. 이러한 요구사항이 만족하지 않는다면, 각 모듈의 실행과 고장이 다른 모듈의 실행결과에 영향을 줄 수 있으며, 애플리케이션의 실행 시간을 예측하기 어렵다[14-16]. 리눅스는 가상 메모리 시스템(Virtual Memory System)을 MMU를 통해 애플리케이션에 제공하여 할당된 메모리 영역 이외의 접근은 제한할 수 있다. 하지만 4GB의 가상 주소범위 안에서 페이지 교체가 발생하여 메모리 할당량이 동적으로 변한다. 또한, 메모리 자원이 고갈되면 아웃오브메모리(OutOfMemory)예외가 발생하여 운영체제는 실행중인 애플리케이션을 종료시켜서 여유 메모리를 확보한다. 따라서 리눅스의 메모리 관리 방법은 ARINC 653표준의 공간분리를 만족하지 않는다.

본 논문에서는 리눅스에 ARINC 653표준의 공간분리 요구사항을 적용하기 위한 기능을 추가하였으며, 동작여부에 대한 실험을 추가적으로 실시하였다. 추가된 기능을 사용하여 초기 파티션 생성 루틴 *startup* 프로세스를 구현했다. 구현 결과를 검증하기 위한 실험환경으로 차세대 우주 임무용 컴퓨터 LEON 4 프로세서를 사용했다. 실험 결과 ARINC 653 표준에서 요구하는 공간분리를 준수하는 것을 확인하였다. 이를 통해 최초로 차세대 우주용 멀티코어 프로세서 LEON 4에서 통합모듈항공전자 시스템을 리눅스를 통해 지원할 수 있는 가능성을 보였다.

본 논문은 다음과 같이 구성되어 있다. 2장에서는 본 논문과 관련된 배경지식에 대해 알아본다. 3장에서는 리눅스의 공간분리의 문제점을 분석한다. 4장에서는 리눅스의 공간분리를 적용방법과 구현을 제안한다. 5장에서는 구현물의 실험결과를 보여준다. 마지막으로 6장에서는 결론과 앞으로 계획에 대하여 기술한다.

II. 배경 지식

1. 통합모듈항공전자 시스템

통합 모듈 항공 컴퓨터 시스템은 하나의 시스템

그림 1. 파티션 개념

Fig. 1 partition concept

실행자(System executive)와 복수의 독립적인 애플리케이션들로 구성되어있다. 각 애플리케이션은 파티션개념을 통해 하드웨어자원을 시간/공간으로 분리해 할당받는다. 프로세서의 관리자/사용자 모드, MMU와 같은 하드웨어 전제 조건은 효율적으로 통합 모듈 항공 컴퓨터 시스템을 구성 할 수 있도록 돕는다. 프로세서의 관리자/사용자 모드를 통해 시스템 실행자와 애플리케이션을 분리 할 수 있으며, MMU를 사용한 가상 메모리 시스템으로 애플리케이션간의 메모리 영역을 분리 할 수 있다. 통합모듈항공전자 시스템은 그림 1과 같이 두 개의 계층으로 구성된다[3].

1) 시스템 실행자 계층:

시스템 실행자는 애플리케이션이 실행될 수 있는 방법을 제공해야 한다. 또한 각 파티션이 서로 영향을 주지 않도록 보장해야 하며 파티션의 스케줄링과 통신 그리고 하드웨어 시그널을 처리할 책임이 있다.

2) 파티션 계층:

공간 분리 (메모리)

공간 분리는 MMU 통한 가상 주소 시스템을 통해 파티션에 할당된 메모리 영역을 보호하며, 다른 메모리 영역의 접근을 제한함으로써 보장한다.

시간 분리 (CPU, I/O)

파티션은 시스템 통합자에 의해 설정된 시간에 따라 고정된 사이클 스케줄링이 지원되어야 한다.

2. LEON4 프로세서의 MMU

효율적인 공간 분리를 지원하기 위해서는 그림 2와 같은 가상 주소 시스템을 MMU통해 지원해야 한다. 가상 주소 시스템을 사용하여 애플리케이션간

그림 2. MMU를 통한 메모리 분리[5]
Fig. 2 Memory isolation with MMU[5]

의 메모리영역 침범을 막기 위해서이다. 만약 MMU 지원 없이 소프트웨어만으로 가상 주소 시스템을 지원하게 된다면 페이징 과정의 처리시간이 길어지기 때문에 시스템의 성능이 떨어지게 된다. LOEN4 프로세서에는 MMU가 포함되어 있으므로 통합모듈 전자 시스템의 공간 분리를 위한 하드웨어 요구사항을 만족한다. LEON4 프로세서는 RISC(Reduced Instruction Set Compute)머신 Sparc V8으로 구성되어 있다. Sparc V8은 3단계 페이징 과정을 통해 가상주소를 물리주소로 연결한다.

III. 리눅스의 공간분리

ARINC 653표준의 공간분리 요구사항은 다음과 같다. 첫째 파티션은 자신에게 할당된 메모리 영역만을 접근할 수 있다. 둘째 파티션은 시스템 통합자에 의하여 설정된 크기의 메모리를 할당받는다. 마지막으로 파티션에게 할당된 메모리는 시스템이 종료될 때 까지 해당 파티션만 사용할 수 있다. 이러한 요구사항은 파티션 애플리케이션의 실행과 고장이 다른 파티션의 실행결과에 영향을 주는 것을 방지하고, 파티션 애플리케이션에 결정적 실행 시간을 보장하기 위함이다[14-16]. LEON4 프로세서는 MMU를 포함하고 있기 때문에 파티션 애플리케이션에 가상 주소 시스템을 제공할 수 있다. 따라서 첫 번째 요구사항인 다른 파티션의 메모리영역 접근 제한을 만족한다. 하지만 설정된 크기의 메모리 할당 요구사항과 할당된 메모리영역에 사용을 보장하는 요구사항을 만족하지 않는다.

1. 메모리 할당량 제한

리눅스는 메모리 관리 정책을 애플리케이션에게 4GB의 가상 주소 기본으로 제공하며, 해당 범위에서 메모리 할당량이 동적으로 변한다. 동적 메모리 관리를 할 경우 애플리케이션에 메모리가 충분히 할당되면 스왑(Swap)이 적게 발생해 좋은 성능을 보인다. 반면 메모리가 적게 할당되면 스왑이 빈번하게 발생하여 낮은 성능을 보인다. 따라서 애플리케이션의 실행 시간을 예측하기 어렵다[15]. 통합 모듈항공전자 시스템은 임무 특성상 실시간 애플리케이션으로 구성되기 때문에 결정적인 실행 시간이 필요하다[1]. 따라서 정적인 메모리 관리를 통해 결정적인 실행시간을 보장해야한다.

리눅스에서 일정 크기의 정적 메모리 할당 요구사항을 만족하기 위해 ARINC 653 표준의 파티션 초기화 과정을 수행하는 *startup 프로세스*를 구현하였다. 파티션 초기화 과정은 시스템 통합자에 의해서 설정된 파티션 자원 할당정보를 읽고 그 내용에 따라 파티션을 생성하는 역할을 한다. 파티션 애플리케이션이 할당된 메모리 이상을 사용하는 것을 제한하기 위해 *rlimit* 구조체를 제어한다. *rlimit* 구조체는 애플리케이션이 할당받을 수 있는 하드웨어 자원을 제한할 수 있는 구조체이다. 그 중 *RLIMIT_AS* 값은 애플리케이션이 사용할 수 있는 가상주소의 크기를 제한할 수 있다. 가상주소의 크기를 제한하게 되면 스레싱이 발생하지 않으면서 최대 사용 물리메모리량을 제한할 수 있다.

2. 메모리 사용의 보장

리눅스는 시스템 전체 메모리 자원을 효율적으로 사용하기 위해 스왑과 아웃오브메모리 예외를 통해 애플리케이션의 메모리를 회수한다. 따라서 애플리케이션에 할당된 메모리량이 동적으로 변한다. 때문에 애플리케이션 실행시간을 예상하기 힘들어지며, 때에 따라 실행이 중단될 수 있다. 하지만 통합모듈항공전자 시스템은 임무 특성상 실시간 애플리케이션들로 구성된다. 때문에 애플리케이션의 결정적인 실행 시간이 필요하며 실행 중에 중단되어서는 안 된다.

메모리 사용의 보장 요구사항을 만족하기 위해서 *mlockall()*와 *task_struct*의 *oomkilladj*를 이용한다. 리눅스가 애플리케이션의 메모리 사용 보장을 위반하는 경우는 다음 두 가지 스왑과 아웃오브메모리 예외이다. 스왑은 보조기억 장치를 주 기억 장치처럼 사용해 실제 주 기억 장치가 크기보다 큰

애플리케이션들을 실행하기 위한 메모리 관리 기법이다. 하지만 스왑 메모리 관리 기법을 적용하면 시스템의 상황에 따라 애플리케이션은 비 결정적인 실행시간을 가진다. 애플리케이션이 모두 스왑 인되어 주 기억 장치와 연결되어 있다면 빠른 실행시간을 보이지만, 메모리 자원 대부분이 스왑 아웃되어 보조 기억 장치에 있다면 자료 접근 시간이 지연되기 때문에 실행 시간이 길어진다. 통합모듈항공전자 시스템 애플리케이션 전체가 주 기억 장치에 연결되어 결정적인 실행 시간을 가져야한다. 파티션 애플리케이션 생성과정에서 *mlockall()*를 사용해 해당 애플리케이션을 주 기억 장치에서 스왑 아웃 되지 않도록 보장한다.

아웃오브메모리 예외는 시스템의 메모리 자원이 고갈되면 발생한다. 메모리 사용량을 바탕으로 종료될 애플리케이션을 선택하며, 운영체제는 선택된 애플리케이션을 강제로 종료해 여유 메모리를 확보한다. 이 과정에서 파티션 애플리케이션이 동작이 중단될 수 있다. 하지만 통합모듈항공전자 시스템은 실시간 애플리케이션을 지원해야 하기 때문에 실행중인 애플리케이션이 중단되어서는 안 된다. 따라서 아웃오브메모리 예외가 발생하더라도 파티션 애플리케이션이 종료되지 않도록 보장해야 한다. 파티션 애플리케이션을 생성하는 과정에서 *task_struct*의 *oomkilladj*변수를 *OOM_DISABLE*로 설정해 아웃오브메모리 예외동작으로부터 파티션 애플리케이션들을 보호한다.

IV. 공간분리 구현

1. 메모리 할당량 제한 구현

본 논문에서 제안하는 ARINC 653 표준의 파티션 초기화 과정을 수행하는 *startup 프로세스*는 시스템 통합자에 의해서 설정된 파티션 자원 할당정보를 읽고 내용에 따라 파티션을 생성하는 역할을 한다. *startup 프로세스*의 구조는 다음 그림 3과 같다. 리눅스에서 파티션을 지원하기 위하여 커널 영역에 ARINC 653 모듈, ARINC 653 스케줄러를 추가 했다. 그리고 사용자 영역에 ARINC 653 표준 인터페이스를 바탕으로 만든 *startup 프로세스*와 APEX(Applcation/Executive) 라이브러리를 구성했다. 커널 영역의 ARINC 653 모듈은 시스템에서 수행될 파티션의 정보를 가지고 있다. ARINC 653 스케줄러는 ARINC 653 모듈에 저장된 파티션의 주기와 실행시간을 바탕으로 실행한다.

그림 3. Startup 프로세스 구조

Fig. 3 Startup 프로세스 struct

사용자영역의 APEX 라이브러리는 ARINC 653 표준 API와 파티션을 지원하기 위한 API로 구성되어 있다. 파티션들과 *startup 프로세스*는 APEX 라이브러리가 제공하는 API를 통해서 ARINC 653 모듈과 통신할 수 있으며, 시스템 통합자에 의해 설정된 파티션 생성, 메모리 할당량, 파티션 주기, 파티션 실행시간, 파티션 이름 등 과 같은 환경설정 데이터를 읽고 파티션의 생성, 환경설정 데이터를 커널 영역의 ARINC 653 모듈로 로딩 하는 역할을 한다. ARINC 653표준에 따르면 모든 파티션은 시스템이 구동될 때 미리 생성된다. 다음과 같은 최소 4가지 정보 메모리 자격요건, 스케줄링 주기, 스케줄링 기간, 입출력포트 속성과 같은 환경 변수들을 시스템 통합자가 시스템이 구동되기 전에 미리 설정하도록 규정하고 있다. 따라서 ARINC 653의 파티션은 시스템이 수행되고 있는 시간에 동적으로 할당이 불가능하며 모든 파티션의 정보와 파티션들의 요구사항을 미리 알 수 있다. 이러한 ARINC 653표준의 요구사항에 맞춰 파티션을 생성하는 *startup 프로세스*가 필요하다. 동작 과정은 다음과 같다.

- 1) 시스템 통합자는 파티션의 환경설정 정보를 작성한다.
- 2) 시스템이 구동하면 *startup 프로세스*는 환경설정 파일을 읽어 파티션정보를 수집한다.
- 3) 획득한 파티션정보를 정의한 APEX 라이브러리를 사용하여 커널 영역의 ARINC 653모듈로 전달한다.
- 4) ARINC 653모듈은 파티션 정보를 확인하고 이상이 없으면 정보를 저장하고 *startup 프로세스*에게 파티션을 생성해도 된다는 값을 전달한다.
- 5) 마지막으로 환경설정에 따라 *RLIMIT_AS*값을 변경해서 파티션 애플리케이션의 메모리 할당량을 제한한다.

그림 4. partitionExecve() 순서도
Fig. 4 partitionExecve() Flow chart

2. 메모리 사용의 보장 구현

파티션 애플리케이션의 메모리 사용을 보장해주는 생성과정 *partitionExecve()*는 시스템 콜로 구현되었으며 매개변수로 실행파일의 경로, 매개변수, 환경변수의 배열의 포인터를 받는다. 반환형은 int 이고 파티션 생성이 성공하면 0을 반환, 실패 할 경우 -1을 반환한다. *partitionExecve()*의 실행 순서는 다음 그림 4와 같다. *startup* 프로세스에서 *partitionExecve()*을 호출하면 Entry.S에 정의한

*sys_partitionExecve()*이 호출이 된다. *sys_partitionExecve()*에서는 실행파일의 경로가 정확한지 확인한 후 정확하지 않다면 -1을 반환하고 종료 하며, 올바른 값이 라면 *do_execve()*를 호출한다. *do_execve()*는 시스템 자원(메모리공간, 실행 프로세스 숫자 제한 값 등)을 확인한 후 생성이 가능하면 파티션에게 자원(CPU,메모리)을 할당한 후 *sys_mlockall()*을 호출한다. *sys_mlockall()*은 *can_do_mlock()*을 통해 현재 리눅스 환경설정이 잠금이 허용이 되는지 *CAP_IPC_LOCK*를 확인한다. 잠금이 허용이 된다면 *do_mlockall()*을 호출한다. *do_mlockall()*은 해당 파티션의 할당된 모든 *vm_area_struct*의 *vm_flags*를 *VM_LOCKED*으로 설정해 해당 가상 주소 메모리를 물리적 메모리에 잠금 한다. 메모리 잠금이 성공적으로 끝났다면 0을 반환하고 *partitionExecve()*의 동작이 끝난다. 위와 같은 *partitionExecve()*의 동작으로 생성된 파티션은 애플리케이션은 할당된 메모리 영역이 시스템에 의해서 회수되지 않는 것이 보장된다.

V. 구현결과 및 성능평가

1. 검증 환경

구현 결과를 검증하기 위한 실험환경으로 차세대 우주 임무용 컴퓨터 LEON4 프로세서를 사용했다. LEON4 프로세서는 Sparc v8를 따르며 200Mhz Dual core CPU에 256 MB DDR2 메모리를 지원한다. 리눅스 2.6.21.1버전을 ARINC 653 표준의 공간분리를 지원하도록 수정했다. 해당 실험 환경에서 메모리 할당량 제한과 메모리 사용의 보장 요구사항을 검증한다.

2. 설정된 크기의 메모리 할당량 제한 검증

애플리케이션에게 정적 크기의 메모리 할당이 가능하도록 구현된 초기 파티션생성 과정인 *startup* 프로세스가 정상적으로 동작하는지 검증한다. 실험 구조는 그림 5과 같다. ARINC 653표준 문서의 파티션 생성예제를 참고했다. 총 4개의 파티션을 생성하고 파티션 1은 2000KB 파티션 2는 4000KB 파티션 3은 6000KB 파티션 4는 8000 KB의 만큼의 메모리를 설정을 한다. 각 파티션은 test라는 동일한 애플리케이션이 동작한다. test는 1byte씩 동적 할당을 실행한다. 더 이상 동적 할당을 실행할 수 없다면 할당을 멈추고 대기한다. 최대 할당량을 통해 환경설정 내용과 같이 메모리가 할당되었는지

그림 5. 메모리 할당량 제한 실험 구조

Fig. 5 Memory assignment limit experiment architecture

표 1. 메모리 할당량 제한 결과

Table 1. Memory allocation limit result

파티션 이름	물리 메모리	가상 메모리
파티션 1	636KB	1972KB
파티션 2	2616KB	3952KB
파티션 3	4596KB	5932KB
파티션 4	6576KB	7912KB

검증한다. 실험 결과는 표 1과 같다.

물리 메모리와 가상 메모리의 차이는 아직 물리 메모리로 연결 되지 않은 코드 영역이다. 환경설정내용과 실제 할당된 가상 메모리간의 크기 차이는 애플리케이션의 동작을 위한 스택영역을 확보하기 때문이다. 스택영역 오차범위 안에서 메모리 환경설정 내용과 같이 메모리 할당량이 제한됨이 검증되었다.

3. 메모리 사용의 보장 검증

파티션 애플리케이션에게 할당된 메모리 영역의 사용을 보장하는 생성과정 *partitionExecve()*를 통해 생성된 파티션 애플리케이션이 시스템의 메모리가 부족한 상태가 되어 아웃오브메모리 예외가 발생해 메모리 회수가 실행되어도 영향을 받지 않음을 검증한다. 실험 구조는 그림 6과 같다. *greed 프로세스*는 기본 리눅스 애플리케이션으로 반복적으로 동적 메모리를 할당해 시스템의 메모리가 부족하게 만들어 아웃오브메모리 예외를 발생하는 역할을 한다. 일반 리눅스의 방법으로 생성한 애플리케이션과 공간 분리가 적용된 리눅스에 의해 생성된 파티션 애플리케이션의 아웃오브메모리 예외동작에서 종료되지 않고 실행이 보장되는 확률을 비교한다. 이와 같은 실험을 파티션 애플리케이션의 메모리 할당량을 달리 하여 20회씩 총 100회 반복 실험 하였다. 그 결과는 그림 7과 같다. 25MB, 50MB, 75MB, 100MB, 125MB로 나누어 메모리를

그림 6. 메모리 사용 보장 실험 구조

Fig. 6 Memory usage protection experiment architecture

그림 7. 메모리 사용 보장 실험 결과

Fig. 7 Memory usage protection result

할당한 결과 *partitionExecve()*를 통해 생성된 파티션 애플리케이션들은 모든 메모리할당량에서 메모리 사용을 보장함을 보인다. 일반 리눅스의 방법을 통해 생성된 애플리케이션의 경우에는 메모리 할당량이 늘어날수록 아웃오브메모리 예외에서 애플리케이션의 메모리 사용 보장률이 낮아짐을 보인다. 그 이유는 메모리 할당량이 클수록 종료해야 할 애플리케이션을 선택될 확률이 높기 때문이다. 125MB를 애플리케이션에 할당한 경우 LEON4의 전체 메모리 250MB의 절반이 되기 때문에 항상 제거 대상이 된다. 본 실험 결과를 통하여 공간 분리가 적용된 리눅스가 파티션의 메모리 사용을 보장함을 검증되었다.

VI. 결론 및 향후 계획

본 논문에서는 차세대 우주용 멀티코어 프로세서 LEON 4 에서 통합 항공 전자 시스템을 리눅스를 통해 지원하기 위해 리눅스에 ARINC 653표준의 공간분리 요구사항을 적용하기 위한 기능을 추가

하였으며, 동작여부에 대한 실험을 추가적으로 실시하였다. 새로운 자료구조와 시스템 콜을 리눅스에 추가했으며 이러한 기능을 사용해 파티션 애플리케이션을 생성하는 초기화 과정을 수행하는 *startup* 프로세서와 *partitionExecve()*를 구현 했다. 실험 결과 ARINC 653 표준에서 요구하는 공간분리를 준수하는 것을 확인하였다. 이를 통해 차세대 우주용 멀티코어 프로세서 LEON 4에서 통합모듈항공전자 시스템을 리눅스를 통해 지원할 수 있는 가능성을 보였다. 하지만 리눅스는 CPU와 I/O자원을 분리하는 시간분리 또한 완전하게 지원되고 있지 않다. 그렇기 때문에 차후 연구를 통해 리눅스에 ARINC 653표준의 시간분리를 구현할 예정이다.

References

- [1] P.J. Prisaznuk, "ARINC 653 role in integrated modular avionics (IMA)," Proceedings of IEEE/AIAA Digital Avionics Systems Conference, 2008.
- [2] J.A. Williams, N.W. Bergmann. "Reconfigurable linux for space flight applications," Proceedings of Military and Aerospace Programmable Logic Devices, pp. 116-128, 2007.
- [3] J. Windsor, K. Hjortnaes, "Time and space partitioning in spacecraft avionics," Proceedings of IEEE International Conference on Space Mission Challenges for Information Technology, pp. 13-20, 2009.
- [4] J. Windsor, M.-H. Deredempt, D.F. Regis, "Integrated modular avionics for spacecraft - User requirements, architecture and role definition," Proceedings of IEEE/AIAA Digital Avionics Systems Conference, pp. 8A6-1-8A6-16, 2011.
- [5] Z. Juan, A. Juan. "Memory Isolation in Many-Core Embedded Systems," Proceedings of International Workshop on High-performance and Real-time Embedded System, 2014.
- [6] N. Kartik, Y.N. Srikant, "Precise shared cache analysis using optimal interference placement." Proceeding of IEEE Real-time and Embedded Technology and Applications Symposium, pp. 125-134, 2014.
- [7] http://www.windriver.com/products/platforms/safety_critical_arinc_653/
- [8] <http://en.wikipedia.org/wiki/XtratuM>
- [9] <http://air.di.fc.ul.pt/air/>
- [10] S. HAN, H.W. JIN, "Kernel-level ARINC 653 partitioning for Linux," Proceedings of the Annual ACM Symposium on Applied Computing, pp. 1632-1637, 2012.
- [11] S. HAN, H.W. JIN, "Full virtualization based ARINC 653 partitioning," Proceedings of IEEE/AIAA Digital Avionics Systems Conference, pp. 7E1-1-7E1-11, 2012.
- [12] J.T. Boyland, "Handling Out of Memory Errors," Proceedings of Workshop on Exception Handling in Object-Oriented Systems, 2005.
- [13] B. Alexander, "MTD Based Compressed Swapping for Embedded Linux," Proceedings of Embedded Linux Conference, 2008.
- [14] G. Lim, C. Min, Y.I. Eom. "Virtual memory partitioning for enhancing application performance in mobile platforms," IEEE Transactions on Consumer Electronics, Vol. 59, No. 4, pp. 786-794, 2013.
- [15] J. Song, X. Zhang, "Adaptive page replacement to protect thrashing in Linux," Proceedings of the Annual Linux Showcase & Conference, 2001.
- [16] B. Oezalp, D. Ferrari, "Two-level replacement decisions in paging stores," IEEE Transactions on Computers, Vol. 100, No. 12, pp. 1151-1159, 1983.

저 자 소 개

김 덕 수

2013년 충남대 정보통신
공학부 컴퓨터 학사.

현재, 충남대학교 컴퓨터
공학과 석사과정.

관심분야: 메모리 최적화,
임베디드 소프트웨어, 항공우
주 시스템

Email: duksoo@cnu.ac.kr

조 현 우

2005년 충남대 정보통신
공학부 컴퓨터 학사.

2007년 충남대 컴퓨터
공학과 석사.

2014년 충남대 컴퓨터
공학과 박사.

현재, 충남대 소프트웨어연구소 전임연구원.

관심분야: 임베디드 시스템 가상화, 항공우주
시스템, 저전력 컴퓨팅

Email: jhwzero@cnu.ac.kr

김 형 신

1990년 한국과학기술원
전산학과 학사.

1991년 University of
Surrey 위성통신학과 석사.

1992~2001년 한국과학
기술원 인공위성연구센터
선임연구원.

2003년 한국과학기술원 전산학과 박사.

2003~2004년 Carnegie Mellon
University 박사후연구원.

현재, 충남대 컴퓨터공학과 교수.

관심분야: 항공우주 시스템, 저전력 컴퓨팅,
임베디드 시스템 소프트웨어.

Email: hyungshin@cnu.ac.kr