멀티미디어 마이크로 커널 M3K에서 프로세스간 통신 구현 및 성능 분석

(An Implementation and Performance Analysis of IPC Mechanism in M3K : A Multimedia Micro-Kernel)

김 영 호 † 고 영 웅 † 아 재 용 † 유 혁 ††
(Young-Ho Kim) (Young-Woong Ko) (Jae-Yong Ah) (Hyuck Yoo)

요 약 최근의 운영체제는 멀티미디어 응용이 보편화되면서 프로세스간 통신에 사용되는 메시지의 크기가 점차 대용량화 되고 있으며, 새로운 하드웨어 플랫폼이 급속히 보급되면서 운영체제의 이식성이 강조되고 있다. 하지만, 기존의 마이크로 커널 구조의 운영체제는 성능 향상을 위하여 마이크로 커널 자체의 이식성을 포기하고 하드웨어 플랫폼에 의존적인 특성을 활용하여 성능을 높이고 있으며, 성능 향상의 주요 대상은 짧은 크기의 메시지를 효율적으로 처리하기 위한 프로세스간 통신 메커니즘이다. 본 논문에서는 다양한 하드웨어 플랫폼 상에서 수행될 수 있도록 이식성을 향상 시킨 M3K(MultiMedia Micro-Kernel) 커널의 프로세스간 통신 메커니즘을 기술하고 있다. 본 논문에서 제시하는 프로세스간 통신 메커니즘은 마이크로 커널의 이식성을 향상시키면서 대용량의 메시지 처리에 있어서 대등한 성능을 보이고 있다.

키워드 : 마이크로커널, M3K, 프로세스간 통신

Abstract As the multimedia application becomes ubiquitous, the size of message used for Inter Process Communication (IPC) grows up to cope with the requirements of multimedia applications. And the rapid development of new hardware platforms makes the portability of operating system more important. But the traditional micro-kernel operating system is implemented platform dependently for better performance, and especially focused on handling short message.

In this paper, we present the design and implementation of IPC mechanism in M3K (MultiMedia Micro-Kernel) to address the above problems. Our IPC mechanism provides enhanced performance and efficiently handles large message without performance degrading.

Key words : microkernel, M3K, Inter Process Communication

1. 서 론

마이크로 커널(Micro-kernel)[1][2][3][4][5][6][7][8][9]은 모노리틱(Monolithic) 커널에 비해서 크기가 작고 커널의 기능성이 사용자 영역에서 수행되는 서버로 구현된다. 따라서 커널 내부에서 발생되는 지연이 작고 예측 가능하므로 실시간 시스템으로 활용되고 있다. 대표적인 예로서 QNX[7]와 같은 시스템을 언급할 수 있으며, 현재 개발중인 Fiasco[10]와 같은 제 2세대 마이크로 커널 운영체제는 실시간 특성을 바탕으로 해서 멀티미디어 시스템으로 확장하고 있다. 일반적으로 마이크로 커널은 하드웨어 초기화 및 프로세스간 통신(Inter Process Communication), 쓰레드 관리(Thread management), 주소 공간 관리(Address space management) 등과 같은 기본적인 기능만을 제공하고, 메모리 관리(Memory management) 및 디바이스 드라이버(Device driver), 파일 시스템(File system), 네트워크 시스템(Network system) 등은 사용자 영역의 서버로 구현하고 있다

· 연구는 한국과학재단의 특정기초 연구과제 연구비 지원(과제번호 97-01-00-09-01-3) 및 한국 전자 통신 연구원의 연구비 지원(과제번호0701-2001-003)에 의한 결과
† 학생회원 : 고려대학교 컴퓨터학과
yhkim@os.korea.ac.kr
yuko@os.korea.ac.kr
jyah@os.korea.ac.kr
†† 종신회원 : 고려대학교 컴퓨터학과
hxy@os.korea.ac.kr
논문접수 : 2001년 2월 8일
심사완료 : 2002년 12월 24일

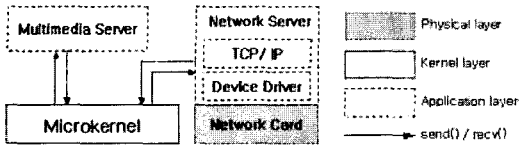

그림 1 마이크로 커널 구조

그림 1에서 보이고 있는 것처럼 마이크로 커널 구조에서는 사용자 영역의 멀티미디어 서버와 네트워크 서버가 마이크로 커널에서 제공하는 프로세스간 통신 메커니즘을 이용하여 메시지를 교환한다. 네트워크 인터페이스를 통하여 데이타가 들어오면 먼저 디바이스 드라이버를 제공하는 네트워크 서버에서 커널 영역으로 데이타를 복사하고, 이후에 그 데이타를 필요로 하는 멀티미디어 서버 영역으로 복사하게 된다. 따라서 커널과 서버 그리고 서버와 서버 사이에는 빈번한 메시지 교환이 발생하게 된다. 그러므로 마이크로 커널 구조에 있어서 프로세스간 통신의 성능은 전체 시스템 성능을 결정짓는 중요한 요소가 된다.

최근에 널리 사용되는 대부분의 운영체제는 멀티미디어를 지원하는 용도로 개발되며, 네트워크 서버를 통해 비디오 프레임과 같이 용량이 큰 데이타를 포함한 패킷(packet)들이 멀티미디어 서버로 전달되기 때문에 대용량의 메시지를 처리하는 프로세스간 통신이 요구된다. 하지만, 지금까지 마이크로 커널의 프로세스간 통신 성능을 최적화 시키기 위한 연구는 대부분 작은 크기의 메시지 처리를 대상으로 하고 있으며, 주로 하드웨어적인 요소를 고려하고 있으며 대표적인 기법으로 짧은 데이타에 대해서 레지스터를 활용하거나 실행되는 명령어의 수를 적게 하여 캐쉬 미스(cache miss)를 줄이는 등 최적화 기법을 사용하고 있다[11]. 그러나 이러한 최적화 기법들은 짧은 메시지에 대해서 어느 정도의 성능 향상을 보여주고 있으나, 대용량의 메시지 처리에 대해서는 성능 향상을 기대하기 어렵고, 하드웨어에 의존적인 특성을 활용하고 있으므로 다양한 플랫폼에 이식하게 될 때 프로세스간 통신 메커니즘이 재설계되어야 하는 문제점을 보이게 된다.

본 논문에서 기술하고 있는 M3K 커널은 언급한 문제점을 해결하기 위하여, 프로세스간 통신을 처리하는 모듈을 하드웨어에 독립적인 컴포넌트 형태로 설계하였으며, 이러한 컴포넌트 구조를 통해서 커널 자체의 이식성(portability)을 높이고 있다. 그리고 커널의 이식성을 향상시키는 구조로 설계한 M3K의 커널 구조가 대용량의 메시지 처리에 있어서 프로세스간 통신의 성능을 저하시키지 않으며, 하드웨어적인 특성을 활용하는 마이크로 커널과 대등한 성능을 보이고 있음을 실험을 통하여 보이고 있다.

본 논문의 구성은 다음과 같다. 2장에서는 관련 연구와 지금까지 프로세스간 통신의 성능 향상을 위한 연구들을 소개한다. 3장에서는 프로세스간 통신의 기반이 될 M3K의 아키텍처에 대해서 설명하고, 4장에서는 M3K의 프로세스간 통신 컴포넌트 구조와 특징에 대해서 살펴본다. 5장에서는 프로세스간 통신 컴포넌트의 구현에 대해서 설명하고, 6장에서는 구현된 프로세스간 통신 컴포넌트의 실험 결과를 분석한다. 그리고 마지막 장에서 결론을 내린다.

2. 관련 연구

커널의 복잡성으로 인한 유지 보수의 어려움을 해결하기 위한 연구가 현재까지 진행되고 있다. 그 중에서도 가장 주목할 만한 것으로는 로더블 모듈(loadable module)과 마이크로 커널을 통한 접근 방법을 들 수 있다.

로더블 모듈은 커널의 일부 기능을 모듈화 하여 커널과 독립적으로 개발하고 런타임(Runtime)에 커널영역으로 모듈을 로딩시켜 모듈의 기능성을 제공하는 모델로서 현재 Solaris와 Linux 등에서 사용되고 있다. 로더블 모듈은 커널에 새로운 기능성을 동적으로 제공할 수 있다는 측면에서 매우 유용하나 그 적용범위가 디바이스 드라이버나 파일 시스템 등으로 제한적이고, 로더블 모듈이 커널영역에서 실행이 되기 때문에 모듈에 오류가 있을 경우에는 시스템에 치명적인 문제를 발생시키게 된다.

마이크로 커널은 운영체제의 기능성을 사용자 영역의 독립된 서버로[7][10] 제공하는 방식이며, Mach[2]가 그 대표적인 예가 될 수 있다. 마이크로 커널은 프로세스나 쓰레드 관리, 프로세스간 통신 그리고 예외 처리 등과 같은 핵심적인 커널 기능만으로 구성한다. 따라서 커널에서는 필수적인 기능만을 제공하고 나머지 기능은 사용자 영역의 서버에서 제공함으로써 커널을 경량화 시킬 수 있다. 그리고 운영체제가 제공하는 서비스를 사용자 영역의 독립적인 서버로 수행시키기 때문에 서버에서 잘못된 수행을 하더라도 다른 서버와 커널에게 치명적인 영향을 주지 않는다. 또한 운영체제의 많은 기능성이 사용자 영역의 서버로 구현될 수 있기 때문에 서버의 개발이 용이하고 운영체제의 기능성을 쉽게 변경할 수 있는 장점을 제공하고 있다. 그러나 마이크로 커널에서는 사용자 영역에서 수행중인 서버들간에 빈번한 프로세스간 통신이 발생되므로, 결국 프로세스간 통신이 전체 시스템 성능에 있어서 병목 지점으로 지적되어 왔다. 따라서 마이크로 커널 구조에 있어서 프로세스간 통신 비용을 최소화 시키는 것이 중요한 과제이며, 다음과 같은 여러 가지 기법이 제시되었다.

첫 번째 방법은 자주 사용되는 운영체제 서비스를 담당

하는 서버를 마이크로 커널 내부에 두는 것으로서 Mach 3.0에서는 AFS 캐쉬 서버를, Amoeba[9]에서는 파일 서버를 커널 내부에 두어서 성능을 향상시키고 있다. 하지만 이러한 방법들은 사용자 영역의 서버를 커널 내부에 둠으로써 커널 구조가 다시 모노리틱화 되는 경향을 보인다.

두 번째 방법은 소프트웨어 및 하드웨어 아키텍처의 특성을 이용한 기법으로서 인터페이스, 알고리즘, 코딩 단계에서 가능한 방법을 모두 사용해서 최적화 시키는 것이다. Fiasco[10]에서는 소프트웨어적인 문맥전환을 이용하여 TLB 미스와 캐쉬(cache) 미스를 줄이고, L3/L4[3]에서는 시스템 콜을 구현할 때 레지스터를 최대한 활용하여 커널과 사용자 영역 사이의 메시지 복사 회수를 줄이는 방법을 사용하고 있다. L3/L4커널에서는 하드웨어 독립성을 다소 포기하고 IPC 성능 향상에 중점을 두어 설계가 되었다. 그러한 예로서 인텔 80486 프로세서 상에서 구현된 L4에서는 세그먼트를 이용하여 하나의 가상 주소 공간에 여러 응용이 동시에 존재 가능하도록 하고 있으며, 이러한 방법은 IPC를 하는 두 응용이 같은 주소 공간에 있다면 주소 공간 변환이 없이 서로간에 통신을 가능하게 한다. 또한 커널에서 지원하는 쓰레드간의 통신이 주로 RPC로 이루어지기 때문에 지연 스케쥴링 기법을 이용하여 쓰레드 큐를 다루는데 있어 캐쉬 실패율을 줄임으로써 IPC 성능을 높이려고 하였다 이러한 방법들은 대개 커널 내부 모듈간의 관계를 긴밀하게 하거나 커널 구조를 특정 하드웨어에 최적화 시키기 때문에, 마이크로 커널 자체의 모듈성이 떨어지고 커널의 호환성 및 이식성도 떨어지게 된다. 또한 최적화 기법들은 대개 작은 크기의 메시지를 처리하는 데는 효과적이지만 멀티미디어 데이타와 같이 메모리 복사 자체로 인한 비용이 큰 경우에는 한계를 갖는다.

세 번째 방법은 Exokernel[1]과 같이 운영체제는 하드웨어 자원을 멀티플렉싱시키기 위한 하위 수준의 작업을 처리하는 최소한의 기능만 가지며, 운영체제의 기능이 라이브러리로 제공되어 응용 프로그램 수준에서 시스템 자원을 관리할 수 있는 방법이 있다. ExoKernel의 성능측정 결과 가장 기본적인 커널 명령어의 경우 Ultix에 비해서 100배 가량 빠르다고 되어있으며, 커널 외부에 존재하는 가상메모리 시스템과 IPC의 경우도 40배 가량 빠르다고 알려져 있다. 하지만. Exokernel에서는 응용프로그램이 시스템을 이용하는 세부적인 메커니즘을 자세히 이해하고 있어야 하며, 시스템 자원을 이용하는데 있어서 정책 결정의 단위가 커널 내부에 있지 않고 응용프로그램에게 있다는 단점이 있다.

3. M3K 아키텍처

M3K[13][14][15]는 멀티미디어를 지원하기 위해 제작된 마이크로 커널로, 하드웨어를 추상화 시킨 코어커널(core kernel)과 이들이 제공하는 인터페이스를 이용하여 실제 커널 서비스를 제공하는 커널 컴포넌트(kernel component)로 이루어 진다. 이 때 하드웨어 특성을 이용한 코어 커널은 코어 커널 인터페이스를 통해서 상위 커널 컴포넌트들에게 추상화 된 서비스를 제공해주고 있다.

그림 2에서는 M3K에서 계층적으로 구분된 코어 커널과 커널 컴포넌트의 관계를 나타내고 있다. 커널 컴포넌트는 코어 커널 인터페이스를 이용해서 하드웨어 독립적인 커널 서비스를 제공하게 된다.

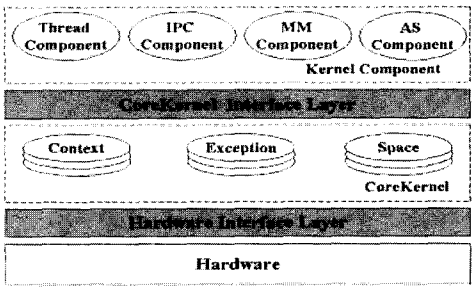

그림 2 M3K 커널 모델

3.1 코어커널

그림 2에서 보는 바와 같이 코어 커널은 실행 문맥, 주소 공간 그리고 예외 상황을 처리하기 위해 각각 문맥(context), 주소(space), 예외(exception) 객체로 이루어져 있다. 문맥 객체는 실행 상태를 저장하는 객체로서 IP(Instruction Pointer), SP(Stack Pointer)와 같은 프로세서의 상태정보를 저장하거나 다른 문맥이 프로세서에 탑재되어 실행될 수 있게 하는 인터페이스를 제공한다. 예외객체는 프로세서의 외부 또는 내부의 예외상황을 표현하는 객체로서 외부 장치로부터의 인터럽트나 실행 도중 발생하게 되는 예외상황을 처리한다. 이러한 예외객체를 사용하기 위해서는 원하는 예외상황에 해당하는 예외객체를 예약하고 예외상황이 발생했을 때 이를 해결하는 핸들러를 등록시켜야 한다. 주소 객체는 주소 공간을 추상화 시킨 객체로서 태스크가 자신의 가상 주소 공간에서 다른 태스크의 방해를 받지 않고 독립적으로 수행될 수 있는 영역을 표현한다.

3.2 커널 컴포넌트

코어커널은 하드웨어의 특성을 이용하여 기본적인 추상화만을 제공하므로 M3K의 기능성을 제공하기 위해서는 마이크로 커널 서비스 정책을 구현한 커널 컴포넌트들의 집합이 필요하다. 이 때 각 컴포넌트는 하드웨어에 상관없이 하위 계층에 있는 코어 커널 인터페이스를 이

용하여 구현 된다. 이러한 커널 컴포넌트에는 주소 공간 컴포넌트(Address Space Component), 메모리 관리 컴포넌트(Memory Management Component), 쓰레드 컴포넌트(Thread Component) 그리고 프로세스간 통신 컴포넌트(IPC Component)가 있다.

다중 주소 공간을 지원하는 M3K에서는 여러 개의 가상 주소공간을 관리하는 기능을 주소 공간 컴포넌트에서 맡고 있다. 이곳에는 가상 주소와 물리 주소와의 매핑(mapping)을 관리하는 페이지 디렉토리와 페이지 테이블에 관한 연산자도 함께 정의하고 있다. 또한 커널 내에서 동적으로 사용되는 메모리를 관리하는 메모리 관리 컴포넌트는 크게 두 가지 방식으로 메모리를 관리한다. 우선 커널이 고정적으로 사용하는 메모리를 제외한 나머지 부분을 버디 시스템(buddy system)[16]으로 관리하면서 페이지 단위로 할당할 수 있다. 그리고 한 페이지 보다 작은 크기의 메모리 요청이 있는 경우, 하나의 페이지를 여러 개의 조각으로 분리해서 관리하게 된다. 즉 M3K에서 페이지 크기보다 작은 메모리 요청이 있는 경우, 미리 생성해 놓은 작은 크기의 메모리 객체들, 중에서 적합한 대상을 선택하여 돌려준다. 이때, 메모리 객체는 16바이트의 배수로 되어 있으며, 메모리 객체 선택 방식은 최적 적합(best-fit) 알고리즘을 사용하고 있다. 즉 메모리 관리 컴포넌트는 요청된 메모리의 크기가 페이지 단위보다 큰 경우는 페이지 블록을 검색하고, 만일 요청된 크기의 페이지 블럭이 없다면, 그 다음 크기(요청된 크기의 두 배)의 블록을 찾아보며, 이 과정은 사용할 수 있는 페이지 블럭을 찾아낼 때까지 계속된다. 만일, 찾아 낸 페이지 블럭이 요청한 크기보다 크다면, 그 페이지 블록은 버디 알고리즘에 의해서 요청한 크기가 될 때까지 분할된다. 그리고 요청된 메모리의 크기가 페이지 단위보다 작은 경우는 미리 분할되어 있는 작은 크기의 메모리 객체 집합에서 최적의 메모리 객체를 선택하게 된다.

앞의 두 컴포넌트는 하드웨어의 특수성에 상관없이 구현되므로 코어 커널 인터페이스를 사용하지 않는다. 그러나 쓰레드 컴포넌트는 코어 커널 인터페이스를 사용하여 쓰레드의 문맥 정보를 관리하게 된다. 쓰레드 관리 정보인 TCB(Thread Control Block)에는 IP, SP와 같이 하드웨어 의존적인 문맥 정보와 프로세스의 식별자, 우선순위(priority)와 같이 하드웨어에 독립적인 정보로 이루어져 있다. 쓰레드 컴포넌트를 구현하는 과정을 살펴보면, 그림 3처럼 먼저 TCB내에서 하드웨어의 의존적인 문맥 정보는 코어 커널에서 제공하는 문맥(Context) 객체를 사용하여 표현하게 된다. 이 때 문맥의 표현뿐만 아니라 쓰레드 간의 문맥 전환도 결국 문맥 객체를 통해 이루어진다. 그리고 수행 시간이나 우선순위와 같이 하드웨어 특수성을 갖지 않는 정보는 쓰레드 컴포넌트에서 새롭게 정의된다. 그러므로 쓰레드 컴포넌트에는 이미 코어 커널에서 생성된 객체를 참조하는 부분과 새로 정의된 부분이 공존하게 된다. 또한 쓰레드 컴포넌트는 실행 가능한 쓰레드 중에서 다음에 실행될 쓰레드를 선택하는 스케줄러를 포함하고 있다. 스케줄러는 현재 실행 가능한 모든 쓰레드 중에서 정해진 정책에 따라 먼저 수행되어져야 하는 쓰레드를 선택한다. 그리고 선택된 쓰레드로의 문맥 전환은 쓰레드 TCB내의 문맥 객체를 이용하게 된다.

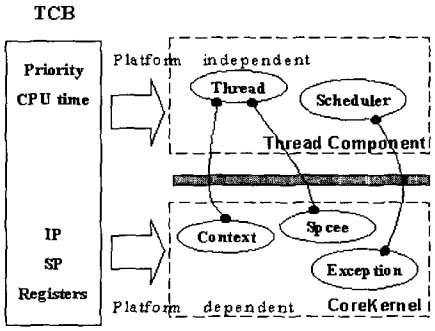

그림 3 M3K 쓰레드 모델

마지막으로 프로세스간 통신 컴포넌트는 서로 다른 주소 공간 및 동일한 주소 공간에서 쓰레드 간의 통신 서비스를 제공한다. 이 때 M3K의 프로세스간 통신 컴포넌트는 프로세스간 통신 송수신에 참여하는 쓰레드 사이에 버퍼를 사용하지 않고 메시지 패싱(message passing) 방식을 사용하며, 이로 인해서 M3K의 프로세스간 통신은 동기적으로 데이타를 주고 받는 특성을 지니게 된다.

4. 프로세스간 통신 컴포넌트 구조

M3KM3K에서 프로세스간 통신 처리 과정은 크게 프로세스간 통신 계층과 시그널(signal) 계층으로 나누어진다. 첫번째 프로세스간 통신 계층은 프로세스간 통신 요청을 처리하기 위해서 필요한 시그널 객체를 생성한 후 시그널 계층으로 전달하는 작업을 한다. 이때 한번의 프로세스간 통신 호출은 여러 번의 시그널 객체를 생성시킬 수 있다. 두 번째 시그널 계층은 우선순위를 갖는 시그널 객체를 대상(target) 쓰레드에 전달하거나, 전달된 시그널 객체를 처리하는 일련의 작업을 수행한다.

프로세스간 통신 컴포넌트에서 시그널 객체를 이용하는 근본적인 이유는 M3K에서 제공되는 프로세스간 통신이 버퍼를 사용하지 않고 동기적으로 수행되기 때문에 프로세스간 통신에 참여하는 쓰레드간에 상태 정보를 알리기 위한 공지(notification) 메커니즘이 필요하기

때문이다. 또한 앞에서도 언급한 바와 같이, 시그널 객체에 우선 순위를 부여함으로써 우선 순위에 따라 프로세스간 통신을 처리할 수 있기 때문이다. 따라서 실시간성을 요구하는 시스템에서 외부 이벤트로 발생된 프로세스간 통신의 응답 시간을 최소화하기 위해서는 이벤트 소스의 우선 순위를 높임으로써 가능해진다.

이러한 시그널 객체에는 시그널 객체의 종류, 시그널 객체를 처리하는 핸들러 그리고 송신 쓰레드에 대한 정보를 포함하고 있다. 이러한 시그널 객체의 전달은 해당 시그널 객체를 송신 쓰레드가 생성해서 수신 쓰레드의 스택에 저장하고함으로써 이루어진다. 그리고 이렇게 전달된 시그널 객체는 수신 쓰레드가 시그널 객체에 포함된 핸들러를 호출함으로써 시그널 객체에 대한 처리를 마치게 된다. 이 때 스택 구조를 사용하는 이유는 외부로부터 전달된 시그널 객체의 우선순위를 유지해서 보다 우선순위가 높은 쓰레드로부터 온 시그널 객체를 처리하기 위해서다.

프로세스간 통신이 수행되는 일련의 처리 과정을 단계별로 살펴보면 다음과 같다.

1. 프로세스간 통신 인터페이스를 호출한다.
2. 호출된 인터페이스에서는 시그널 객체를 생성(①)하고 이를 대상 쓰레드의 시그널 스택에 저장(②)한다.
3. 대상 쓰레드로 문맥을 전환(③)한다.
4. 문맥이 전환된 대상 쓰레드는 자신의 시그널 스택에서 가장 위에 있는 시그널 객체를 가져온다.(④)
5. 스택에서 인출된 시그널 객체 내에 있는 핸들러를 호출(⑤)한다. 이 때 시그널 객체의 핸들러는 임계영역 내에서 처리되므로 한 쓰레드가 동시에 서로 다른 시그널 객체를 처리할 수 없다.
6. 방금 처리된 쓰레드 보다 우선 순위가 높은 시그널 객체가 스택에 존재하는지 조사한다. 만약 우선 순위가 높은 시그널 객체가 스택에 존재하면 위의 4번 과정부터 반복한다.
7. 프로세스간 통신을 호출한 쓰레드로 문맥 전환(⑥)을 한다.

이와 같이 프로세스간 통신 컴포넌트는 프로세스간 통신 계층과 시그널 계층을 포함하고 있으며 쓰레드 정보를 관리하기 위해서 쓰레드 컴포넌트에서 제공하는 인터페이스를 사용하게 된다. 그러므로 다른 시스템으로 M3K를 이식하더라도 하드웨어와 독립적으로 구현된 프로세스간 통신 컴포넌트를 통해서 프로세스간 통신 기능을 제공할 수 있다.

그림 4는 프로세스간 통신이 수행되는 과정을 보여주고 있다. 그림에서 알 수 있듯이 하나의 프로세스간 통신을 완전히 수행하기 위해서는 최소한 두 번의 문맥전환이 이루어진다. 따라서 쓰레드간의 프로세스간 통신은 수행된 프로세스간 통신의 수보다 많은 문맥 전환을 초래하게 되

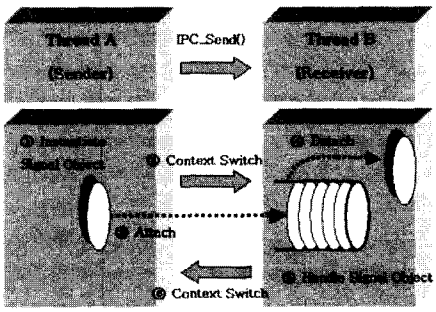

그림 4 프로세스간 통신 처리 과정

고, 전체 시스템 성능의 병목 지점으로 지적될 수 있다.

이 후 실험에서는 실제로 프로세스간 통신을 수행하는 동안 발생되는 문맥 전환의 비용이 전체 성능의 병목 지점이 되는가를 살펴보고, 프로세스간 통신을 통해 전달되는 메시지의 크기에 따라 어느 부분이 병목지점으로 나타나는지를 분석한다.

5. 프로세스간 통신 컴포넌트의 구현

M3K는 C++ 언어와 어셈블리 언어를 이용하여 구현하였다. 특히 하드웨어 의존적인 코어 커널부분은 C++와 인라인(inline) 어셈블리 언어를 사용하였으며, 커널 컴포넌트 부분은 모두 C++로 구현하여 하드웨어 독립성을 갖게 하였다. 현재 M3K는 코어 커널을 구성하는 문맥, 주소, 예외 객체를 구현하였고 이를 기반으로 메모리 관리를 위한 메모리 관리 컴포넌트와 쓰레드 관리를 위한 쓰레드 컴포넌트 그리고 쓰레드 간의 통신 서비스를 제공하기 위한 프로세스간 통신 컴포넌트를 구현하여 커널을 구성하였다.

이렇게 구성된 M3K의 커널 이미지는 작성된 부트로더(boot loader)를 이용하여 시스템 부팅시 메모리에 로딩되고 코어 커널과 커널 컴포넌트를 초기화한 후에 커널 서비스를 제공한다. 프로세스간 통신 컴포넌트의 M3K의 구성하는 하나의 커널 컴포넌트로 다음과 같은 방법을 통해서 구현하였다.

5.1 시그널 객체의 생성

```
class Signal {
public:
    virtual handler ( );
}
class mySignal : public Signal {
public :
    virtual handler ( ) ;
}
```

그림 5 시그널 객체의 의사코드

그림 5는 M3K에서 객체 지향 언어(C++)로 구현된 시그널 객체의 모습을 간략화해서 나타낸 것이다.

위의 의사코드에서 mySignal 클래스는 시그널을 생성할 객체로서 handler라는 virtual 함수를 갖는 Signal 클래스를 상속한다. 따라서 Signal 클래스를 상속 받는 모든 클래스는 시그널이 갖는 속성을 그대로 지니게 된다. 또한 각 시그널 객체의 역할과 특징을 반영하기 위해서 확장된 Signal 클래스에서는 handler 함수를 오버라이딩(overriding)하게 된다. 예를 들어, 메시지를 전달하는 시그널 객체를 정의하기 위해서는 Signal 클래스를 상속하는 클래스를 정의하고 전달할 메시지를 복사하는 handler를 정의하게 된다.

5.2 시그널 객체의 전달

이렇게 정의된 시그널 객체를 실제로 다른 쓰레드에게 보내기 위해서는, 먼저 해당 시그널 객체의 클래스로부터 객체를 생성하고 sendSignal 함수를 이용하게 된다. sendSignal 함수는 전달하려는 시그널 객체를 대상(target) 쓰레드의 시그널 스택에 저장하고 대상 쓰레드로 문맥 전환하게 된다. 대상 쓰레드는 문맥이 전환되자마자 시그널 스택에 저장된 시그널 객체를 처리한다. 이 때 대상 쓰레드가 다른 쓰레드로부터 온 시그널 객체를 처리하던 중이었으면 보낸 시그널 객체는 바로 처리되지 못하고, 스택에 저장만 되었다가 나중에 처리된다.

5.3 시그널 객체 핸들링

시그널 객체를 받은 쓰레드가 시그널 객체를 처리하는 과정을 의사코드로 작성하면 그림 6과 같다. 우선 시그널 객체를 처리하기 전에 이미 다른 쓰레드로부터 온 시그널 객체를 처리하던 중이었는지 확인하기 위해서 현재 쓰레드가 가지고 있는 세마포어(semaphore)를 점검한다. 이 때 이미 다른 쓰레드로부터 온 시그널 객체를 처리하던 중이었으면 세마포어에 의해 설정된 임계영역으로 인해 처리가 끝날 때까지 대기하게 된다. 그렇지 않으면 세마포어를 이용하여 임계영역으로 설정하고

```
if(lock ( ) )
    return FALSE ;
while (signal=Stack.pop ( ) ) {
    signal→handler ( ) ; // call signal's handler
    sender=siganl→sender ( ) ;
    first=Stack.top ( ) ; // get top in the stack
    if(first&&(first→priority>sender→priority))
      continue ; // if higher priorty exist in the stack
    if(sender→priority>this.priority)
      switch_to(sender) ;
}
unlock ( ) ;
return TRUE ;
```

그림 6 시그널 객체 핸들링 의사코드

스택에 저장된 시그널 객체를 하나씩 처리한다. 시그널 객체에 대한 처리는 스택 맨 위에 있는 시그널 객체부터 빼서 그 시그널 객체에 포함된 핸들러를 호출함으로써 이루어진다. 이 때 핸들러는 virtual 함수로 지정이 되어 있기 때문에 실제 시그널 객체의 종류에 따라 다른 핸들러가 호출된다.

스택의 맨 위에서 꺼낸 시그널 객체의 처리가 끝나면 현재 시그널 객체와 시그널 스택 맨 위에 남아 있는 시그널 객체의 우선 순위를 비교한다. 이 때 시그널 객체의 우선 순위는 그 시그널을 보낸 쓰레드의 우선 순위로 결정된다. 만일 현재 처리된 시그널 객체의 우선 순위보다 스택 맨 위에 저장된 시그널 객체의 우선 순위가 높으면 스택에 있는 시그널 객체부터 처리한다. 그리고 현재 처리된 시그널 객체의 우선 순위가 스택 맨 위에 남아 있는 시그널 객체보다 높은 경우에는 처리된 시그널 객체의 송신 쓰레드로 문맥 전환이 이루어 진다.

5.4 프로세스간 통신 메시지 송신

그림 7은 이렇게 구현된 시그널 객체를 이용해서 실제 프로세스간 통신 메시지를 보내는 doSend 함수를 보여주고 있다.

```
doSend(Thread target, IPCMsg message){
        target->enqueue(this);
        signal = new msgSignal (message);
        ret=sendSignal(target, signal);
        while(ret===NOT_READY){
            sendSignal(target, singal);
        }
}
```

그림 7 시그널을 전달하는 doSend 함수

doSend 함수는 보내려고 하는 메시지와 수신 쓰레드에 대한 참조를 함수의 매개 변수로 받는다. 메시지를 다른 쓰레드로 보내는 경우, 송신 쓰레드는 우선 프로세스간 통신 상대인 수신 쓰레드의 대기 큐에 송신 쓰레드의 참조를 삽입하게 된다. 그리고 보내고자 하는 메시지를 생성된 Signal 객체에 포장(wrap)한 후에 sendSignal 함수를 통해서 수신 쓰레드로 전달한다.

생성된 시그널 객체를 수신 쓰레드로 보내는 순간에 메시지를 받는 수신 쓰레드가 준비되지 않은 경우, 즉 방금 보낸 메시지를 수신 쓰레드가 아직 요청하지 않은 경우에는 이미 만들어진 시그널을 계속해서 대상 쓰레드에게로 보내게 된다. 이 때 송신 쓰레드는 sendSignal 함수의 리턴 값을 통해서 수신 쓰레드의 준비 상태를 알 수 있다. 왜냐하면 수신 쓰레드는 요구하지 않은 메시지에 대한 시그널 객체를 받게 되면 시그널 핸들러를 통해서 NOT_READY라는 결과값을 보내기 때문이다.

하지만 doSend 함수가 불리기 전에 수신 쓰레드가 먼저 메시지를 요청했다면 송신 쓰레드의 sendSignal 함수를 통해 시그널 객체에 포함된 메시지는 수신 쓰레드에게 바로 전달된다. 만약 송신 쓰레드가 수신 쓰레드보다 먼저 프로세스간 통신 함수를 호출하게 되면, 수신 쓰레드의 프로세스간 통신 큐에는 송신 쓰레드가 삽입되고 송신 쓰레드는 수신 쓰레드가 메시지를 요청할 때까지 대기하게 된다.

5.5 프로세스간 통신 메시지 수신

그림 8은 메시지를 받기 위한 doReceive 함수로 doSend 함수와 동일한 타입의 매개 변수를 갖는다. 하지만 각 매개 변수는 doSend 함수와 정반대의 용도로 사용된다. 우선 sender 변수는 메시지를 보낸 쓰레드를 가리키는 것으로, NULL이 아닌 경우에는 특정 쓰레드로부터 메시지를 받을 때 사용된다. 그리고 message 변수는 받은 메시지를 참조하기 위한 것으로 프로세스간 통신이 성공적으로 끝나면 송신 쓰레드에서 보낸 메시지, 즉 송신 쓰레드에서 보낸 시그널 객체로부터 추출(unwrap)된 메시지를 가리키게 된다.

```
doReceive(Thread sender, IPCMsg message){
        if(sender is not ready){
            signal = new wakeupSignal();
            sendSignal(sender, signal);
        }
        sleep(); //sleep until message is received
}
```

그림 8 메시지를 받기 위한 doReceive 함수

doReceive 함수에서는 메시지를 보낸 송신 쓰레드가 프로세스간 통신 함수를 호출했는지의 여부를 먼저 판단하기 위해서 쓰레드 내에 있는 프로세스간 통신 큐를 확인해야 한다. 송신 쓰레드가 먼저 호출한 경우에는 큐에 존재하는 송신 쓰레드의 참조를 이용해서 수신 쓰레드가 현재 수신가능 상태에 있음을 송신 쓰레드에게 알리게 된다. 이를 위해서 수신 쓰레드는 WakeupSignal 시그널 객체를 생성하고 송신 쓰레드에게 전달하게 된다. 지금까지 수신 쓰레드의 프로세스간 통신 요청을 기다린 송신 쓰레드는 수신 쓰레드가 보낸 WakeupSignal를 받고 메시지 전달을 시도하게 된다. 수신 쓰레드의 문맥으로 처리되는 시그널 객체의 핸들러에서는 송신 쓰레드를 수신 쓰레드 프로세스간 통신 큐에서 인출하고, 시그널 객체에 포함된 메시지를 수신 쓰레드로 복사하게 된다.

위의 설명에서도 알 수 있듯이, 수신 쓰레드가 먼저 doReceive 함수를 호출하고 난 뒤에 송신 쓰레드의 doSend 함수 호출하는 것이 가장 이상적인 상황이라 할 수 있다. 그렇지 않고 doSend 함수가 먼저 호출되면 수신 쓰레드가 송신 쓰레드에게 수신 가능 상태를 알리기 위한 시그널 객체를 더 보내야 하기 때문에 전체적인 프로세스간 통신 오버헤드가 더 커지게 된다.

6 실험 및 성능 평가

본 장에서는 M3K의 프로세스간 통신 컴포넌트의 성능을 세밀히 분석하였으며, 프로세스간 통신의 각 구간에서 차지하는 지연 시간의 비율을 보이고 있다. 실험 결과에서 하드웨어적인 특성을 활용한 프로세스간 통신의 최적화 방식은 메시지의 크기가 커짐에 따라서 성능에 미치는 영향이 거의 없어짐을 보이고 있다. 또한 호환성과 이식성을 강조한 M3K와 하드웨어 특징을 활용한 독일의 Fiasco 마이크로 커널과 프로세스간 통신 성능을 상호 비교함으로써, M3K의 커널 구조가 시스템의 성능을 저하시키는 직접적인 요소가 아님을 보이고자 한다.

6.1 M3K의 프로세스간 통신 지연 시간 분석

본 장에서 M3K의 프로세스간 통신 컴포넌트가 수행되는 경로를 따라 각 구간에서 걸리는 시간을 측정하고, 메시지의 크기에 따라 각 구간이 전체 시간에서 차지하는 비율을 분석하였다. 또한 실험에 사용되는 하드웨어의 사양을 달리하여 실험을 함으로, 프로세스간 통신 수행의 병목 지점이 어떠한 하드웨어적 특성에 의해서 영향을 받는지 살펴보고 있다. 본 논문에서는 인텔(Intel) 계열의 펜티움(Pentium) 프로세서를 대상으로 실험하였으며, 세밀한 시간 측정을 위해서 펜티움 프로세서에 내장되어 있는 TSC(Time Stamp Counter)를 사용하였다.

표 1은 펜티움 II-233MHZ에서 M3K의 프로세스간 통신 컴포넌트를 이용하여 두 쓰레드 간에 메시지를 전달할 때, 프로세스간 통신 컴포넌트 내에서 호출되는 경로를 구간 별로 측정한 TSC 값을 나타낸다. 측정 구간은 그림 4에서 설명한 프로세스간 통신 단계를 기본으로 해서 7개로 세분화하여 정의하고 있다.

구간 1(T1)은 송신 쓰레드에서 프로세스간 통신 계층의 doSend함수가 호출된 후, 사용자 영역의 데이타를 커널 영역으로 복사하고 시그널 객체를 생성하는 데까지 걸린 시간이다. 구간2(T2)는 시그널 객체를 송신 쓰레드가 수신 쓰레드에게 전달하는 데 걸린 시간이다. 구간 3(T3)은 송신 쓰레드에서 수신 쓰레드로 문맥이 전환되는 데 걸린 시간을 나타낸다. 구간 4(T4)는 문맥 전환된 수신 쓰레드의 시그널 스택에서 시그널 객체를 인출하는 데 걸린 시간이다. 구간 5(T5)는 시그널 스택에서 인출된 시그널 객체를 처리하는 데 걸린 시간으로서 메시지를 커널 영역에서 수신 쓰레드의 사용자 영역으로 복사하는 데 걸린 시간이 된다. 구간 6(T6)은 시

그널 객체를 처리한 수신 쓰레드에서 다시 송신 쓰레드로 문맥을 전환하는 데 걸린 시간이다. 마지막으로 구간 7(T7)은 송신 쓰레드가 메시지를 성공적으로 보내고 마무리하는 시간이 된다.

표 1 구간별 수행 시간 측정

(단위 : TSC)

	32B	64B	128B	256B	512B	1KB
T1	821	1265	2184	4251	7866	15038
T2	44	47	47	44	44	43
T3	183	180	178	179	179	215
T4	159	158	154	153	153	169
T5	395	490	717	3809	7407	14605
T6	192	192	192	192	192	192
T7	567	571	565	564	560	560

그림 9는 표 1의 데이타를 그래프로 표시한 것이며, 수행 시간이 미미한 T2, T3, T4, T6, T7의 값을 합하여 other로 표시하였다. 그림 9에서 알 수 있듯이, 메시지의 크기가 커짐에 따라 구간 1과 구간 5의 비중이 점점 커지고 있으며, 나머지 구간은 거의 변화가 없음을 알 수 있다. 메시지의 크기가 32Byte일 때 T1과 T5의 수행 시간이 전체 수행 시간에서 각각 35%, 17%를 차지하고 있었으나, 메시지의 크기가 1KByte인 지점에서는 T1과 T5의 수행 시간 비율이 각각 49%, 47%로 전체 메시지 전송 시간의 대부분을 차지하게 된다.

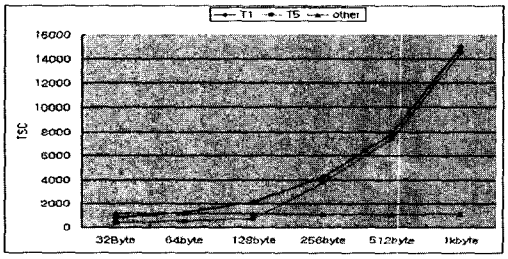

그림 9 구간별 성능

반면, T1과 T5를 제외한 나머지 구간 other에서는 메시지의 크기가 증가하더라도 아주 미묘한 차이만을 보이고 있다. 앞에서 구간을 나눌 때 알 수 있듯이, T1과 T5가 갖는 공통점은 각 구간에서 일어나는 작업의 대부분이 다른 영역의 메모리 주소로 메시지를 복사한다는 점이다. 다시 말해서 두 구간에서는 공통적으로 메모리 복사 작업을 수행하고 있다. 따라서 메시지가 더욱 커질수록 메모리 복사에 따른 비용은 프로세스간 통신 컴포넌트의 성능을 좌우하는 직접적인 원인이 된다.

다음은 메모리 복사에 영향을 미치는 요소를 알아보기 위해서 다음과 같은 조건에서 실험하였다. 실험의 인자로는 CPU 코어 속도(CPU core speed)와 외부 버스 속도(External bus speed)를 사용하고 있다.

표 2 실험에 사용된 하드웨어 특성

CPU model	CPU core speed	External bus speed
P-100	100MHZ	50MHZ
PII-233	233MHZ	66MHZ
PII-300	300MHZ	100MHZ
PIII-500	500MHZ	100MHZ

그림 10은 표 2의 실험 환경에서 메시지 크기에 따른 프로세스간 통신 수행시간 측정값을 그래프로 표현하였다. 그림 10에서 보이고 있는 것처럼 메시지의 크기가 커짐에 따라 모든 프로세서마다 프로세스간 통신 수행시간 곡선이 상승하는 모습을 보이고 있다. 그러나 Pentium II-300와 Pentium III-500의 경우에는 CPU 코어 속도 차이에도 불구하고 앞의 두 프로세스와 달리 비슷한 양상을 갖는 곡선이 나타난다. 이렇듯 프로세스간 통신 수행시간에 결정적인 영향을 주는 것은 CPU 코어 속도가 아닌 외부 버스의 속도이며, 이는 하드웨어인 외부 버스 구조가 메모리 복사의 성능을 좌우한다는 것을 알 수 있다. 따라서 그림 10에서 Pentium-100, Pentium II-233 그리고 Pentium II-300이 나타내는 현격한 프로세스간 통신 수행시간의 차이는 근본적으로 상이한 외부 버스 속도에 따른 결과로 해석할 수 있다.

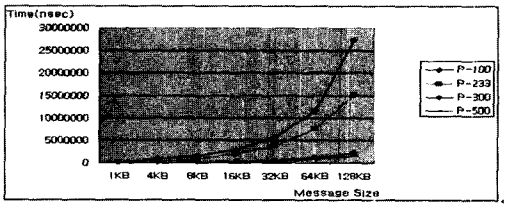

그림 10 프로세스간 통신 수행 시간

6.2 Fiasco 마이크로 커널과 M3K의 프로세스간 통신 시간 비교

본 장에서는 대표적인 제 2세대 마이크로 커널로 알려진 Fiasco 커널과의 프로세스간 통신 시간을 비교하고 있다. Fiasco 마이크로 커널은 L3/L4 마이크로 커널을 C++ 언어를 사용하여 재 작성한 것이며, L3/L4가 가지고 있는 대부분의 특징을 포함하고 있다. L3/L4 마

이크로 커널은 짧은 메시지에 대해서 레지스터를 활용하고 있으며, 커널 내부 구조를 최적화 하고 커널 크기를 최소화 하여 프로세스간 통신 시간을 극도로 줄여 놓았다. 그림 11은 Fiasco와 M3K의 전체 프로세스간 통신 수행시간 값을 보여주고 있다.

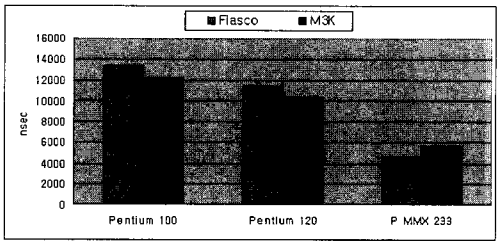

그림 11 M3K와 Fiasco의 프로세스간 통신 성능 비교

그림 11에서 알 수 있듯이 프로세스간 통신 시간이 거의 비슷하게 나타나고 있으며, 하드웨어 사양과는 거의 무관하게 비슷한 성능을 보이고 있다. 그림 12는 Fiasco와 M3K 의 프로세스간 통신 구간별 수행시간을 나타내고 있다. Fiasco 커널과 M3K 커널의 프로세스간 통신 모듈이 서로 상이하지만, 메시지 전달 방식이라는 공통성을 가지고 있으므로 Fiasco의 메시지가 처리되는 각 경로를 추적하여 M3K의 구간과 유사한 부분으로 분할하여 측정을 하였다.

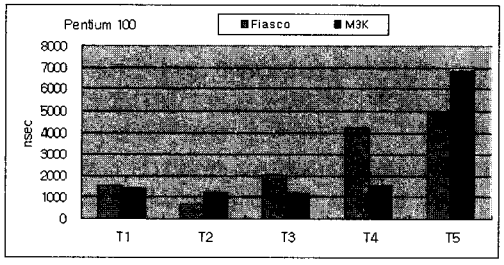

그림 12 M3K와 Fiasco의 구간별 성능 측정값

그림 12를 보면, T1과 T2, T3구간의 경우 거의 차이가 없는 수행시간을 보여주고 있지만, T4구간의 경우에는 Fiasco의 수행시간이 M3K에 비해서 대략 3배 가량 길어진 것을 볼 수 있다. T4에서 수행 되는 작업은 송신 쓰레드를 확인하고 관리하는 것과 실제 데이타 복사를 처리하는 부분이다. 이때 프로세스간 통신 메시지를 처리하기 위해서 Fiasco의 경우 레지스터를 이용하여 데이타를 복사하기 때문에 메모리 복사를 하는 M3K보다 작은 수행시간이 걸린다. 하지만, 송신 쓰레드를 확인하고 관리하는 부분에 있어서 Fiasco는 메시지 큐를 두고 큐를 통해서 송신 쓰레드를 확인 및 관리하는 반면, M3K는 메시지의 헤더만 읽어서 확인하고 관리하는 방법을 택하기 때문에 상대적으로 단순한 구조를 취하고 있다. 따라서 실험 결과상으로 하드웨어의 특징을 활용하는 Fiasco에 비해서 단순한 메시지 관리 구조를 취하는 방식이 프로세스간 통신 비용을 더욱 크게 줄이는 요인이 되고 있음을 알 수 있다.

7. 결 론

본 논문에서는 다양한 플랫폼에 이식하기 쉽게 설계된 M3K 커널에서 제공하고 있는 프로세스간 통신 컴포넌트의 특징과 구현에 대해서 살펴보았으며, 실험을 통하여 성능을 분석하였다. 본 논문은 마이크로 커널의 프로세스간 통신의 성능을 향상 시키려는 노력으로 하드웨어의 특성을 활용하는 방법이 어느 정도의 성능을 개선하는데 도움을 줄 수 있으나, 새로운 플랫폼으로 이식하는데 많은 비용을 발생시키고 있음을 지적하고 있다. 또한 하드웨어 플랫폼에 의존적인 마이크로 커널의 설계 방법은 멀티미디어 응용과 같이 대용량의 프로세스간 통신을 필요로 하는 새로운 운영체제의 요구에 있어서 성능상 큰 효용성이 없음을 보이고 있으며, 이를 실험을 통하여 보이고 있다.

본 논문에서는 프로세스간 통신이 수행되는 경로를 여러 개의 구간으로 나누고 메시지 크기에 따라 각 구간이 전체 소요 시간에서 차지하는 비중을 분석하였으며, 실험 결과 메시지의 크기가 커질수록 메모리 복사가 프로세스간 통신의 병목 지점임을 알 수 있었다. 결과적으로 대용량의 메시지 처리에 있어서 하드웨어의 특징을 활용하는 방법은 거의 도움이 되고 있지 않음을 보였다. 또한 메모리 복사의 성능을 결정하는 가장 큰 요인이 외부 버스 속도이며, 따라서 외부 버스 속도가 빠른 하드웨어를 사용하지 않고서는 대용량의 메시지 처리를 하는데 한계가 있음을 알 수 있다. 마지막 실험에서는 M3K 커널과 제 2세대 마이크로 커널의 대표적인 사례인 Fiasco 커널과의 직접적인 프로세스간 통신 수행 시간을 비교해보았으며, 실험 결과에서 볼 수 있듯이 성능상 거의 차이가 없음을 알 수 있다. 이는 이식성 향상을 목적으로 개발된 M3K 커널의 프로세스간 통신 메커니즘이 성능을 향상 시키기 위하여 다양한 하드웨어 특징을 활용하고 있는 Fiasco와 비교하여 성능 차이의 직접적인 원인이 아님을 보이는 것이다.

향후 연구 방향은 이식성을 향상시키는 구조로 설계되어 있는 M3K의 프로세스간 통신 메커니즘을 확장하는 것이며, 이를 위하여 비동기식 메시지 패싱에 대한 연구 및 대용량의 메시지 전달을 효율적으로 처리할 수 있는 페이지 공유 기법등에 대해서 연구를 진행하고 있다.

참고문헌

[1] D.R. Engler, M.F. Kaashoek, and J. OToole Jr, "Exokernel and operating system architecture for application-specific resource management," in Proceedings of the 15th ACM Symposium on Operating Systems Principles, pp. 251-266, December 1995.

[2] M. Acceta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, M. Young, "MACH: A New Kernel Foundation for UNIX Development," in Proceedings of USENIX, summer 1986.

[3] Jochen Liedtke, "On microkernel construction" in Proceedings of the 15th ACM Symposium Operating System Principle(SOSP) (Copper Mountain Report, Colo., Dec. 1995). ACM Press, New York, 1995. pp 237-250.

[4] OS Group, TU Dresden, IBDR, The Dresden Real Time Operating System Project. Online at http://os.inf.tu-dresden.de/project/.

[5] Brian Bershad, Craig Chambers, Susan Eggers, "SPIN an extensible microkernel for application-specific operating system services," Technical Report 94-03-03, Dept. of Comp. Sci. and Eng., University of Washington, Seattle, February 1994.

[6] H. Tokuda, T. Nakajima, and P. Rao, "Real-Time Mach: Towards a Predictable Real-Time System," in Proceedings of USENIX 1st Mach Workshop, October 1990.

[7] Dan Hildebrand, "An Architectural Overview of QNX," in 1st USENIX Workshop on Micro-kernels and Other Kernel Architectures, pp. 113-126, Seattle, WA, April 1992.

[8] Yasuhiko Yokote, "The Apertos reflective operating system : The concept and its implementation," in OOPSLA92 Conference Proceedings, 1992.

[9] A.S. Tanenbaum, M.F. Kaashoek, R.van Renesse, and H. Bal, "The Amoeba Distributed Operating System A Status Report," Computer Communications, vol. 14, pp. 324-335, July/Aug 1991.

[10] http://os.inf.tu-dresden.de/fiasco/

[11] Jochen Liedtke, Kevin Elphinstone, Sebastian Schönberg, Hermann Härtig, "Achieved IPC Performance," in HotOS, May 1997.

[12] 양순섭, 고영웅, 조유근, 신현식, 최진영, 유혁, "컴포넌트 기반 커널을 위한 프레임워크," 춘계 정보과학회 학술대회 논문집, 1999.

[13] 김영호, 고영웅, 유혁, "M3K에서의 쓰레드 컴포넌트 구현," 추계 정보과학회 학술대회 논문집, 1999.

[14] 김영호, 고영웅, 유혁, "M3K에서 프로세스간 통신 컴포넌트 설계 및 구현," 추계 정보과학회 학술대회 논문집, 2000.

[15] 김영호, 고영웅, 아재용, 유혁, "M3K에서 프로세스간 통신 구조 및 성능평가," 컴퓨터시스템 연구회 논문집 2000.

[16] Uresh Vahalia, *UNIX Internals : the new frontiers*, Prentice Hall, Englewood Cliffs, NJ, 1996.

김 영 호

1999년 고려대학교 컴퓨터학과(학사)
2001년 고려대학교 컴퓨터학과(석사)
2001년 ~ 현재 (주)오프너스 연구원
관심분야는 운영체제, 실시간 시스템, 내장형 시스템

고 영 웅

1997년 고려대학교 컴퓨터학과(학사)
1999년 고려대학교 컴퓨터학과(석사)
1999년 ~ 현재 고려대학교 컴퓨터학과 박사과정
관심분야는 운영체제, 실시간 시스템, 멀티미디어

아 재 용

2000년 고려대학교 컴퓨터학과(학사)
2000년 ~ 현재 고려대학교 컴퓨터학과 석사과정
관심분야는 운영체제, 보안 운영체제, 네트워크

유 혁

1982년 서울대학교 전자공학과 학사
1984년 서울대학교 전자공학과 석사
1986년 University of Michigan 전산학 석사
1990년 University of Michigan 전산학 박사
1990년 ~ 1995년 Sun Microsystems Lab
1995년 ~ 현재 고려대학교 이과대학 컴퓨터학과 부교수
관심분야는 운영체제, 멀티미디어, 네트워크