# 다중개성 운영체계의 구현에 관한 연구 : 직접 프로시져 호출 방식의 통신기법

조 시 훈[†]·방 남 석[†]·이 준 원[††]

## 요 약

다중개성을 지원하는 운영체계는 각 사용자의 필요성에 따라 수정과 확장이 용이하게 이루어져야만 한다. 마이크로커널 구조의 운영체계가 다중 개성 운영체계로서 적합한 반면, 프로세스간 통신(interprocess communication: IPC)에서 발생되는 마이크로커널 구조의 부담때문에 시스템 성능이 저하되는 문제점을 갖고 있다. 본 논문에서는 기존의 IPC 기법 성능을 개선하기 위하여 운영체계의 구성요소들간에 직접 프로시져 호출방식의 새로운 IPC 기법을 제안한다. 새로운 기법에 의한 통신 부담은 최상의 경우에는 로컬 프로시져 호출 수준이고, 최악의 경우에도 기존의 메시지 전송 통신기법보다는 성능이 향상된다.

# An Implementation of an Operating System with Multiple Personalities : design of the Direct Procedure Call IPC scheme

Shihoon Cho[†]·Namseok Bang[†]·Joonwon Lee[††]

## ABSTRACT

An operating system should be easy to extend and to be customized to accomodate the diverse needs when it is to support multiple personalities. Though microkernel is the best OS architecture for such customization, it raises performance issues due to its intrinsic overhead in interprocess communication. In this paper, we present a new IPC mechanism that overcomes the traditional IPC performance lag by providing a direct bridge between processes. The communication overhead is similar to the one of local procedure call, which is the known minimal overhead. Even in its worst case, our scheme is shown to perform better than the other alternative, the message passing scheme.

## 1. 서 론

다중개성 운영체계는 동일한 시스템 구조상에서 다양한 기능의 운영체계가 제공되는 것으로서 사용자 수준의 응용 분야가 가지는 다양성에 따라 각 응용 분야의 특별한 목적에 맞게끔 수정과 확장이 용이하게 지원되는 운영체계를 의미한다. 다중개성 운영체계를 구현하기 위해 기존의 운영체계가 가지고 있는 한계점을 극복하기 위한 노력은 1)UNIX[1]와 같은 모노리딕(monolithic)커널을 확장하여 응용분야의 요구사항을 만족시켜주는 방법과 2)마이크로커널(Microkernel)이라는 새로운 구조의 운영체계를 개발하는 방법으로 구분되어진다. 전자의 경우 복잡한 커널구조에 의하여 지속적

※ 이 논문은 1994년 한국학술진흥재단의 공모과제 연구비에 의하여 연구되었음.
† 준 회 원 : 한국과학기술원 전산학과
†† 정 회 원 : 한국과학기술원 전산학과 교수
논문접수 : 1997년 12월 8일, 심사완료 : 1998년 6월 22일

1) UNIX is a trademark of AT&T

인 확장성, 이식성, 그리고 신뢰성을 보장하기 위한 많은 노력이 요구되는 반면에, 후자는 간결한 인터페이스와 운영체계의 각 기능이 수평적 구조로 구성되어 시스템 확장의 용이성 및 보다 유연성 있는 시스템 구성을 가능하게 한다. 따라서, 사용자 수준의 다양한 목적에 따라 보다 쉽게 운영체계의 구조를 변경하거나 새로운 기능을 추가하기 위해서 다중개성 운영체계는 결국 운영체계의 개발과 확장면에서 더욱더 유리한 마이크로커널 구조 방식을 택할 것으로 추측된다. 뿐만 아니라, 다중개성 운영체계 구현을 위한 노력은 마이크로커널 구조의 운영체계에서 나타나는 여러 문제점을 극복하기 위한 방법들로 귀결되어진다.

마이크로커널 구조의 운영체계에서는 기존의 커널이 담당했던 많은 기능들을 사용자 수준의 프로그램(서버)들이 수행하고 마이크로커널은 서버사이의 메시지 전송과 일반적인 기능만을 담당한다. 하지만 현실적으로는 보안을 고려하여 많은 서버가 커널 혹은 준커널 수준에서 기능을 수행한다. 사용자가 요구한 서비스를 처리하기 위해서는 사용자와 서버, 서버와 서버, 혹은 서버와 커널 사이의 인터페이스가 이루어져야 하는데 대부분의 마이크로커널 운영체계는 메시지 전송(message passing)을 이용한 프로세스간 통신기법(IPC:Inter Process Communication)을 사용하고 있다. 그러나, 일반적인 메시지 전송을 통한 IPC를 수행할 때 수반되는 많은 사건들 -메시지 복사, 문맥 교환(context switch), 커널으로의 트랩(trap), 접근 확인(access validation), 서버의 스케줄링 등- 은 IPC 부담(overhead)으로서 작용하고 결국은 시스템 성능저하와 비효율성을 초래한다.

본 논문에서는 메시지 전송 IPC 기법의 부담을 줄이고 마이크로커널 구조의 다중 개성 운영체계 성능향상을 위한 새로운 IPC 기법인 직접 프로시져 호출방식(DPC: Direct Procedure Call)의 IPC를 제안한다. 제안된 DPC IPC기법은 운영체계의 모든 구성요소들 - 서버들과 커널 - 을 하나의 가상 주소 공간(virtual address space)에 위치시킨 후, 임의의 한 서버(프로세스)에서 다른 서버로 요청한 서비스를 로컬 프로시져 호출과 유사한 개념을 이용하여 커널로 하여금 동일한 주소 공간에 위치하는 다른 서버의 프로그램을 호출하는 것이다. 프로세스간의 통신에 필요한 여러 가지 사항을 로컬 프로시져 호출의 수준으로 유지함으로써 메시지 전송 IPC 기법에서 발생되는 불필요한 부담을 줄이며 시스템 성능 향상을 가져오도록 한다.

본 논문의 구성은 다음과 같다. 2장에서는 기존의 연구사항을 살펴보고, 3장에서는 DPC IPC 기법 및 설계요건들을 살펴본다. 4장에서는 새로운 DPC IPC 기법을 다른 기법들과 비교하여 성능 평가를 수행한다. 그리고 끝으로 5장에서 결론을 맺는다.

## 2. 관련 연구

프로세스간의 통신에 필요한 메시지 전송 IPC 때문에 발생되는 운영체계의 성능저하를 극복하기 위한 노력은 Mach 2.5/3.0[1,15], LRPC[3] 등에서 보여지지만 이들 시스템들은 현격한 성능 향상을 제공하지는 않는다.

Carnegie-Mellon 대학서 개발된 Mach는 다양한 메시지 전송방식을 제공하며 메시지의 종류에 따라 메시지 포맷을 달리 구성하여 전송한다. Mach에서의 IPC 방식은 한 번의 시스템 호출에 의하여 메시지를 전송하게 되는데, 메시지를 전송하기 위해서 커널의 트랩과 메시지의 복사 등이 수반되므로 실질적으로 일반적인 IPC 메카니즘과 동일한 부담을 가지며 이에 따른 성능 감소를 야기한다.

Utah 대학에서 연구된 [10]에서는 Mach의 IPC에서 발생되는 커널과 사용자 사이의 모드 전환에 따른 문제점을 줄이기 위해서 사용자 수준의 제한된 서버(conformant server)를 booting시에 선택적으로 커널내에 상주시키므로써 불필요한 부담을 제거하고자 하였다. 따라서, 사용자와 서버사이의 상호작용의 경우에는 DPC IPC 기법처럼 사용자가 원하는 서비스를 커널 수준에서 제공하므로써 메시지 복사, 커널/사용자 entry/exit 회수가 감소되므로 시스템 성능 향상의 결과를 가져온다. 하지만, 커널 내부에서 커널과 서버 혹은 서버들 사이의 상호작용에서는 문제점을 가진다. Mach의 RPC 기법에서는 사용자가 요청한 서비스를 수행하기 위해서 해당 서버들은 항상 커널로의 트랩을 발생하고 필요한 RPC stub을 생성해야만 하며, 커널의 주소 공간을 공유하는 서버들에 대해서 최소한의 부담을 가지는 문맥교환이 일어나게 된다. 또한, Mach의 RPC 기법에서는 프로세스 migration에 따른 부담을 초래한다. 사용자가 서비스를 해당 서버에 요청하는 경우에 사용자 수준의 thread에서 서버의 thread로 migration이 발생되므로 이를 위한 상태 보존을 PCB에 하게 된다.

그런데, 서버에서 서비스를 수행하면서 필요한 기능을 다른 서버에 요청하는 경우에는 커널로의 트랩을 통한 해당 서버로 전환되므로 이전 서버의 상태를 보존해야 할 필요성이 발생한다. 원래의 PCB에는 이미 사용자 수준의 thread에 대한 상태가 보존되고 있으므로 이를 위해서 새로운 구조가 PCB에 존재하여야 한다.

Lightweight Remote Procedure Call(LRPC)은 동일한 기계에서 수행되는 프로세스 영역(domain)사이의 통신을 위해 고안된 기법이다. LRPC는 일반적인 통신 기법이 아니고 다중프로세서 환경하에서 RPC[2]을 사용한 IPC 기법의 한 형태로서 제공된다. 프로세스간 통신이 서로 다른 기계(cross-machine) 수준이 아닌 동일한 기계 위의 프로세스 사이(cross-domain)에서 발생되는 RPC에 대해서 이를 최적화시키므로써 기계간 RPC에서 발생되는 부담 - stub 프로시져 생성, 메시지 버퍼 확보, 접근 확인, 서버 스케쥴링, 문맥 교환, dispatch 등 - 을 줄이는 것을 목적으로 한다. 즉 메시지 복사의 회수를 줄이기 위해 동일 기계 위의 모든 프로세스간에 공유되어지는 메시지 버퍼를 확보하고, 메시지 버퍼를 통해 프로세스간의 메시지를 전달한다. 클라이언트(혹은 사용자)로부터 메시지를 궁극적으로 전달하는 커널으로의 메시지 복사는 이루어지지 않지만 커널으로의 트랩과 서버의 접근 확인, 문맥 교환, 서버의 스케쥴링 등의 부담은 존재하므로 메시지 전송 IPC 기법과 유사한 부담을 가지게 된다.

두 프로세스 사이의 메시지 복사를 가장 빨리 수행하기 위해서는 역설적으로 메시지를 전혀 이동하지 않으면 된다. 현재의 대부분 컴퓨터시스템은 공유 메모리(shared memory)의 개념을 지원한다. 공유 메모리를 사용한 프로세스간의 메시지 전달방식은 메시지 전송 IPC를 통한 것보다 빠르게 수행될 수 있다. 메시지를 전달하는 과정에서 필요한 메시지 큐의 부담은 없으며 커널은 공유 메모리의 보호만 관여하면 된다. 또한 서버와 커널사이의 메시지 복사가 필요없게 된다. 하지만 공유 메모리의 접근을 동기화(synchronization)하기 위한 수단 - 세마포어, 모니터 등 -을 제공하여야 한다. 예를 들어 공유 메모리에 쓰기 작업을 하는 중에 다른 프로세스의 읽기 작업이 허용되면 잘못된 메시지의 공유가 발생할 수 있다. 공유 메모리를 통한 메시지 전달방법에서는 메시지 전달에 앞서 공유 메모리 접근 허가를 얻어야 하는 IPC 부담이 존재하게 된다.

## 3. DPC IPC 기법 및 설계 요건들

### 3.1 DPC IPC 기법

DPC IPC 기법을 지원하기 위해 시스템 모형(그림 1)에서 필요한 사항은 다음 세 가지로 구분되어진다. 첫째, 단일 주소 공간을 이용한 운영체계의 구성이다. 기존의 메시지 전송 IPC 기법에서는 프로세스간의 통신 혹은 사용자가 요청한 서비스의 수행을 위해서는 각 프로세스가 가지는 주소 공간사이의 이동이 필요한데 이는 1장에서 나열한 문맥교환, 커널로의 트랩, 메시지 복사 등의 부담을 발생시킨다. 각 프로세스 고유의 주소 공간 사이에 제어(control)를 이동함으로써 초래되는 위와 같은 부담을 줄이기 위해서 DPC IPC 기법에서는 커널을 포함하여 운영체계를 구성하는 모든 프로세스들(서버들)을 하나의 가상 주소 공간하에 유지하고 주소 공간 이동으로 인한 불필요한 사건을 줄이므로써 성능 향상을 추구하고자 한다. 따라서 커널과 독립적으로 유지되는 프로세스들을 단일 주소 공간으로 만들기 위한 해결책이 제안되어야 하며 각 서버 프로세스들에 할당된 가상 주소 공간들을 하나의 가상 주소 공간으로 재조정하기 위한 매커니즘이 필요하게 된다. 둘째, 임의의 한 서버 프로세스에서 요청할 수 있는 IPC 서비스들에 대해 해당 프로시져들의 상태 정보나 시작 주소 등 정보 유지가 필요하다. 이를 위해서 DPC IPC 기법에서는 마이크로커널과 독립적으로 유지되는 서버들이 수행하는 프로시져의 종류 및 시작 주소에 대한 정보를 loader에 의해서 전달받아서 동적으로 마이크로커널에서 유지한다(ipc_table). 셋째, IPC 서비스를 요청한 서버 프로세스(Initiator)와 서비스를 제공하는 서버 프로세스(Responder) 사이의 연결(매핑) 매커니즘의 지원이 요구된다. 단일 주소 공간하에서 임의의 한 서버 프로세스에서 요청한 IPC 서비스가 동일 문맥하에서 수행되어지기 위해서는 실제로 서비스를 제공하는 서버 프로세스와의 매핑이 이루어져야 된다. DPC IPC 기법에서는 두 가지 종류의 프로시져들- stub_ipc_exe 와 ipc_exe_을 이용한 동적 바인딩을 통해 지원한다. 마이크로커널에 위치한 ipc_exe는 Initiator와 Reponder사이의 실질적인 동적 바인딩을 수행

2) 일반적으로 RPC를 사용하는 IPC 기법은 네트웍의 저수준에서는 궁극적으로 메시지 전송을 사용한 IPC를 이용하므로 본 논문에서는 RPC IPC 기법을 메시지 전송 기법을 사용한 IPC와 같은 의미로 가정한다.

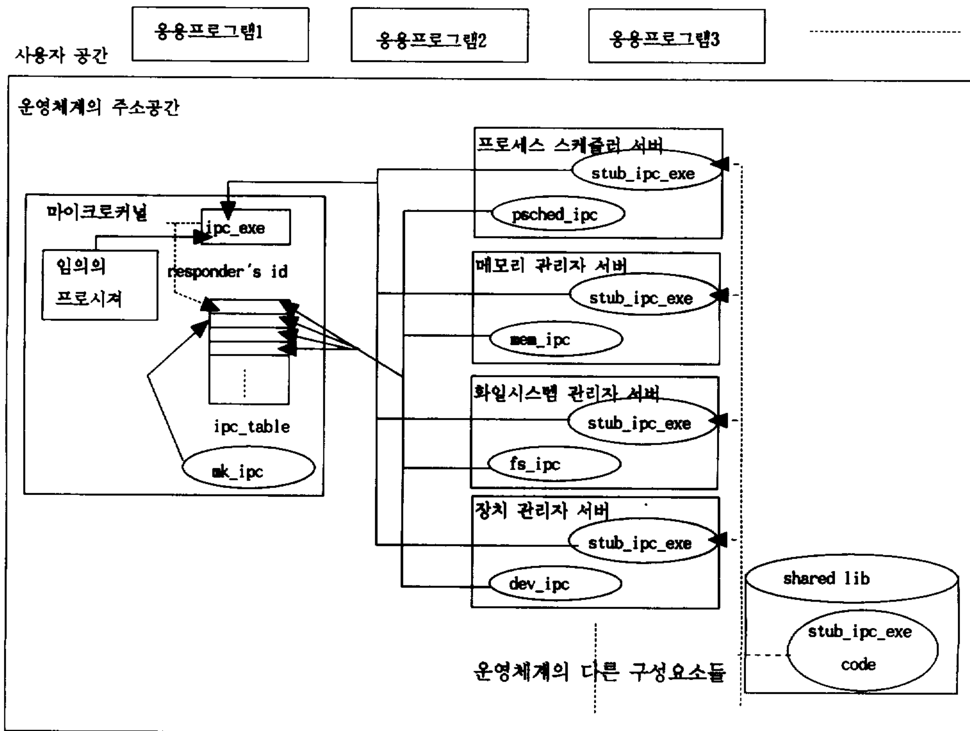
하는 프로시저이며, 각 서버 프로세스에 있는 stub_ipc_exe는 IPC 서비스를 요청하는 기능을 담당하게 된다.

### 3.2 단일 주소 공간을 이용한 운영체계 구성

일반적으로 시스템 서버 프로그램들은 컴파일러에 의해 독립적으로 생성되어지고 시스템에 종속적인 실행 화일 형태를 가지게 된다. 또한 각 서버들은 운영체계로부터 제공받은 자신만의 가상 주소 공간을 사용하게 된다. 3.1절에서 살펴보았듯이 DPC IPC 기법에서는 커널과 모든 서버들을 하나의 단일 주소 공간에 위치시키는 것을 목적으로 하므로 UNIX와 같은 주소 공간 할당 방법으로서는 단일 주소 공간을 생성하는데 많은 어려움이 따르며 실행 화일의 크기가 상당할 것으로 추측된다. 따라서, 새로운 DPC IPC 방식에서는 정적인 linker에 의해 생성되는 실행 화일의 크기를 가능한 한 줄이며 궁극적으로 단일 주소 공간에 모든 서버들의 화일 이미지를 할당할 수 있도록 각 서버에 할당되는 세그멘트의 크기를 작게 유지하고 세그멘트내의 각 region들을 연속되게 위치시킨다. 이를 위해서 다음 사항들을 가정한다.

- 세그멘트화 페이징(segmented paging) 가상 메모리 시스템을 지원한다.
- linker에 전달하는 각 region의 시작 주소는 가상 주소 공간의 연속적인 세그멘트의 주소이다.
- linker는 가상 주소 공간의 작은 영역만을 사용하는 서버 프로그램을 생성한다.

DPC IPC 기법에서는 서버들에 할당된 가상 주소 공간들을 하나의 정적인 가상 주소 공간으로 조정하기 위해서 다음 두 가지 방법을 제안한다. 첫번째는, 실행 화일을 재생성한다. Loader가 서버들을 적재할 때, 서버에 할당된 세그멘트의 각 region의 이미지가 새로운 시작 주소를 갖도록 강제적으로 다시 링크하는 것으로서 다른 어떤 방법보다 시스템 성능을 개선시킬 수 있다. 이 방법의 선결 조건으로서는 재할당(relocation)정보가 정적 linking후에도 보존되어야 한다. 두번째는




(그림 1) DPC IPC 기법을 지원하는 시스템 모형
(Fig.1) System Model for supporting a DPC IPC Scheme

CPU의 주소 방식(addressing mode)을 사용하는 것이다. 할당된 세그멘트의 시작 주소를 하드웨어의 기본값으로 특정 레지스트(base register)에 유지한 뒤 서버가 실행될 때 이 값을 더해서 가상 주소의 새로운 값으로 참조하도록 하는 방법이다. 후자의 경우, 컴파일러가 CPU 레지스터들을 다른 목적으로써 사용해서는 안되며, 참조하고자 하는 가상 주소를 기본 레지스터의 값을 더해서 간접적으로 알아내야 하므로 전자보다는 성능이 떨어지는 단점을 가진다.

운영체계의 단일 주소 공간으로 할당된 독립적인 서버들을 위해 공통적으로 사용되는 하나의 페이지 테이블을 지원한다. 시스템 로딩시에 각 서버들의 주소를 페이지 테이블에 첨가하므로써 페이지 테이블을 구성하고 마이크로커널의 관리하에 다른 서버들이 공통적으로 페이지 테이블에 접근하도록 한다.

### 3.3 서버 프로세스사이의 동적 바인딩

DPC IPC에서 각 서버들은 다른 서버로부터 요청되어지는 서비스 즉 IPC 사건들을 처리하기 위해 서비스에 해당되는 프로시저를 가진다. 예를 들어 메모리 관리자는 mem_ipc 프로시저를 가지며, 이 프로시저는 메모리 관리자를 필요로 하는 다른 서버들로부터 불려진다. 각 서버들이 보유하는 프로시저들은 커널이 유지하는 자료 구조 - IPC 프로시저 테이블(ipc_table)- 에 정보가 저장되며 부팅시에 마이크로커널 이미지에 적재되어진다(그림 2). Ipc_table에 있는 각 프로시저 이름은 서버들의 역할을 의미하는 접두어 - 마이크로 커널(1), 프로세스 스케쥴러(2), 메모리 관리자(3) 등 - 에 의해 구별되어지며 동적으로 테이블에 첨가된다[2,7]. 예를 들어 프로세스 스케쥴링을 담당하는 psched_ipc 프로시저의 주소를 테이블에 첨가시킨다고 가정하자. 우선 Loader에 의해 프로세스 스케쥴링 서버를 메모리에 적재한 후 psched_ipc의 프로시저의 주소를 알아낸다. 마이크로커널내의 ipc_table에서 서버 인덱스 값이 프로세스 스케쥴링인 필드에 위의 주소 값을 링크시키므로써 테이블의 값을 할당하게 된다.

IPC 서비스를 요청하는 클라이언트와 사건을 수행하는 서버사이의 매핑은 마이크로커널이 유지하는 ipc_table의 정보에 의해서 동적 바인딩(dynamically binding)을 통해 이루어진다. 즉, IPC 서비스를 요청하는 Initiator와 서비스를 실제로 제공하는 Responder사이

| 서버 프로세스의 ipc 프로시저 주소 | 상태 플래그 |
|---|---|
| * (mk_ipc) () | active |
| * (psched_ipc) () | active |
| * (mem_ipc) () | active |
| * (fs_ipc) () | active |
| * (dev_ipc) () | active |
| * (comm_ipc) () | active |
| . | . |
| . | . |

(그림 2) IPC 프로시저 테이블(ipc__table)
(Fig. 2) IPC procedure table

의 동적 바인딩은 ipc_exe 프로시저에 의해 이루어진다. Ipc_exe 프로시저는 두 가지 종류가 존재한다. 마이크로 커널을 제외한 모든 서버에 존재하는 stub_ipc_exe 프로시저는 공유 라이브러리에서 host 라이브러리와 동일한 기능을 수행하는 것으로서 서비스를 요청하는 서버에 의해 수행되며, 마이크로커널내에 존재하는 ipc_exe 프로시저는 바인딩을 실제로 수행하는 프로시저로서 공유 라이브러리의 target 라이브러리의 기능과 유사하다[20]. 동적 바인딩 과정을 살펴보면 아래와 같다(그림 3).

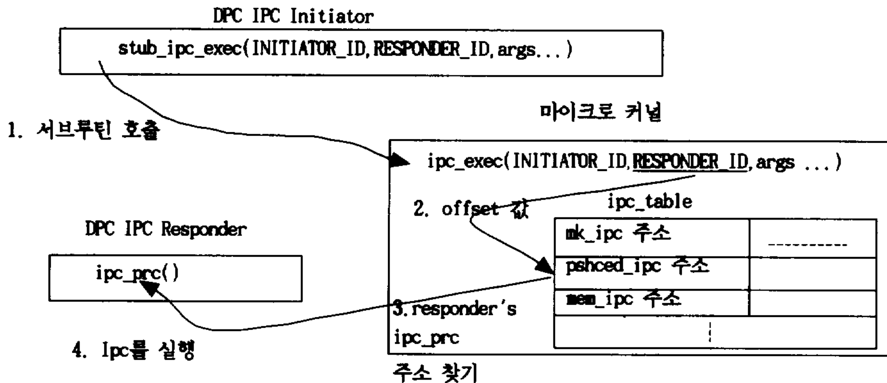
1. 서비스를 필요로 하는 IPC Initiator에 의해 stub_ipc_exe가 호출된다.
2. 호출된 stub_ipc_exe 프로시저는 동적으로 링크된 마이크로커널내의 ipc_exe를 호출한다.
3. Ipc_exe 프로시저는 Initiator로부터 넘겨받은 인자 중에서 서비스를 제공할 Responder의 종류를 마이크로 커널내에 있는 ipc_table에서 찾아낸다.
4. Responder의 프로시저를 실제로 수행한다.

여러 서버들이 동일한 서비스를 요청하는 경우에, 즉 다중 IPC Initiator들이 동시에 하나의 Responder를 요청하게 되면 프로시저들에 의해 전역 데이터가 수정될 가능성이 있게 된다. 이를 방지하기 위해서 critical section을 지정하고 lock/unlock 방식에 의해 전역 데이터를 보호한다.

### 3.4 DPC 운영체계의 특성

#### 3.4.1 Ipc_prc 에 의해서 사용되어지는 스택의 분류

Ipc_prc 프로시저에 의해서 사용되어지는 스택의 분류는 DPC IPC Initiator와 IPC 서비스를 요청한 주체에 따라서 크게 세 가지로 나누어진다. 첫째, 마이크로 커널이 DPC IPC Initiator이며 사용자 응용프로그램에




(그림 3) DPC IPC에서 프로시저 수행 과정
(Fig. 3) Processing DPC IPC

의해서 IPC 서비스가 제기되었을 때는 일반적인 시스템 호출과 마찬가지로 사용자 응용프로그램의 커널 스택을 사용하게 된다. 따라서 Responder의 ipc_prc 프로시저는 커널 스택을 사용한다. 둘째, 마이크로커널이 DPC IPC Initiator이면서 또한 IPC 서비스 - 페이지 부재 -를 제기한 경우이다. 페이지 부재 처리기가 사용하는 스택은 시스템에 종속적이지만 일반적으로 인터럽트된 사용자 응용 프로그램이 사용하던 커널 스택을 사용하게 된다. 마지막으로, 서버 프로세스가 DPC IPC Initiator인 경우에 서버 프로세스는 자신의 스택을 사용한다.

### 3.4.2 DPC IPC 구조에서 서비스 요청 방식

DPC IPC 구조에서 서비스를 요청하는 방식은 크게 IMMEDIATE와 DELAYED 두 가지로 분류된다. Initiator가 IPC 서비스를 IMMEDIATE 방식으로 요청하는 경우에 Responder의 ipc_prc는 자신의 프로시저 수행이 완료되기 전까지는 실행 제어를 자신을 호출한 클라이언트 Initiator에게 되돌려주지 않는다. 서비스를 요청하는 Initiator가 현재 가지고 있는 작업이 없거나 바쁘지 않는 경우가 이에 해당된다. 반면에 DELAYED 방식으로 서비스를 요청한 경우, 서버의 ipc_prc는 요청된 서비스를 포맷팅한 후 자신의 지연 작업 큐(delayed-job-queue)에 첨가한다. 그리고 즉각적으로 제어를 Initiator에게 되돌려 준다. IPC Initiator가 무척 바쁜 작업을 수행중이거나 예외 발생 등의 예상치 못한 사건이 일어나서 Responder가 즉각 수행을 끝내야 할 때가 이에 해당된다. 예외적으로 서버들의 마이크로커널에 대하여 DELAYED 방식에 의한 서비스 요청은 허용되지 않는다. 왜냐하면, 마이크로커널은 큐에 쌓인 사건들을 주기적으로 검사하는 프로그램 코드를 clock 인터럽트 처리 프로시저에 첨가해야만 되므로 지연된 사건은 결국 인터럽트 모드하에서 처리되어지기 때문이다.

### 3.4.3 지연된 서비스의 수행

DELAYED 방식으로 서비스를 요청하면 ipc_prc는 REQUEST_MODE 인자에 의해 이를 파악하고 지연될 사건을 저장하기 위한 메모리를 할당하며, 나중에 다시 실행되기 위해서 필요한 정보를 저장하는 dlydevt 라는 구조체를 만든다. 각 서버에 존재하는 dlydevt_q에 dlydevt를 첨가하고 DELAYED_EVT_SUCCESS의 값을 Initiator에게 즉각 되돌려 준다. Ipc_exe는 서버에 대한 wake_up 플래그를 설정하고 나중에 서버가 다시 스케줄되면 지연된 서비스를 큐로부터 꺼내어서 수행시킨다.

### 3.4.4 서버 프로그램의 실행 모드

DELAYED 방식으로 서비스가 요청된 경우에 서버(Responder)는 wake_up된 후에 독립적인 프로세스로서 자신의 문맥에서 사건을 처리한다. 따라서 메시지 복사와 기계 상태의 변화 등의 부담을 가지고 실행 속도가 느려진다. 반면에 IMMEDIATE 방식에 의해 서비스를 수행할 경우에는 Responder의 해당 프로시저를 클라이언트의 Initiator에게 전달하게 되므로써 서버는 독립적인 프로세스로서의 의미보다는 다른 서버 문맥의 일부분으로서 수행되어진다. IPC 서비스를 처리하

는데 전자와 같은 부담은 없게 된다.

3.4.5 서버를 위한 문맥교환

일반적으로 서버의 프로세스들이 스케줄링될 경우에는 서버의 상태를 보존하거나 복원하기 위해 문맥교환이 일어나게 된다. 특히 MMU 레지스터의 값들을 적합하게 유지시켜야만 된다. 그러나, DPC IPC의 경우 모든 서버들이 하나의 페이지 테이블을 공유하므로 페이지 테이블의 정보들이 supervisor 모드에서 MMU 레지스터에 저장되고 가상 주소 공간들이 MMU 레지스터들로부터 값들을 가지므로, 서버가 스케줄되는 것에 관계없이 MMU 레지스터의 문맥을 보존/복원할 필요성이 없게 된다.

## 4. 성능 평가

### 4.1 성능 평가 방법

본 논문에서의 성능 평가는 각 IPC 방식들의 절대적인 성능을 측정하기보다는 IPC 방식들의 실행 속도의 상대적인 비교에 초점을 둔다. 특히 DPC IPC 방식의 성능을 측정하기 위해 1장에서 언급된 IPC 부담의 여러 구성 요소들의 발생 빈도를 비교하고 분석하는 이론적이고 간접적인 방식을 선택한다.

### 4.2 IPC 부담의 분석

각각의 IPC 방식들이 가지는 부담은 운영체계의 구조 혹은 운영체계가 제공하는 IPC 방식에 의존한다. 다시 말해서 임의의 한 시스템은 특정한 운영체계의 구조적인 특성으로부터 야기되는 IPC 부담을 가질 수 있다. 또한 동일한 IPC 부담을 가지는 두 가지 운영체계에서, 부담의 정도는 내부적으로 IPC 방식을 지원하는 운영체계의 방법에 의하여 차이를 보일 수 있다.

표 1에 나타난 각 IPC 기법들의 상대적인 성능 평가 결과와 앞 절에서 기술한 사항으로부터 Immediate 방식의 DPC IPC 기법이 다른 IPC 기법들보다 성능이 우수함을 알 수 있다. 또한, 비록 Delayed 방식의 DPC IPC 기법에서 IPC 부담은 존재하지만 다른 IPC 기법들에 비해 처리 시간이 상대적으로 작으므로 DPC IPC 기법이 마이크로커널 구조의 시스템에서 성능향상을 위한 개선 방법이 될 수 있음을 보여준다.

### 4.3 상대적인 성능 평가

4.3.1 환경

IPC 방식들의 성능을 평가하기 위해 IPC 성능에 영향을 줄 수 있는 다른 요인은 배제한다. 우리는 다음 세 가지 요구조건을 만족하는 마이크로커널 구조의 시스템에서 모든 IPC 방식들을 실행하는 것으로 가정한다.

- 동등한 하드웨어 플랫폼의 선택 : 각각의 IPC 방식이 서로 다른 하드웨어에서 실행된다면 올바른 IPC 성능 평가를 얻을 수가 없게 된다. 따라서 IPC 성능 평가에 영향을 줄 수 있는 하드웨어 요소를 배제한다.

〈표 1〉 IPC 기법들의 상대적인 성능평가
〈Table 1〉 Relative performance evaluation for IPC Schemes

| 부담 \ IPC 기법 | Message Passing | LRPC | Shared Memory | DPC | |
|---|---|---|---|---|---|
| | | | | immediate | delayed |
| IPC 데이터 복사 | E | E | E | NE | E |
| 커널로의 트랩 | E | E | NE | NE | NE |
| 사용자와 커널공간 사이의 복사 | E | NE | NE | NE | NE |
| 접근 확인 | E | E | E(high) | E | E |
| 스케줄링 | E | E | E | NE | E |
| 문맥교환 | R | R | R | NE | PR |
| RPC Stub | NE | E | NE | NE | NE |

(E: 존재한다. NE: 존재하지 않는다. R: 반드시 요구한다. PR: 부분적으로 요구한다.)

〈표 2〉 IPC 기법들의 매개변수 값
〈Table 2〉 Parameters of IPC Schemes

| 부담/매개변수 \ IPC 기법 | | Message Passing | LRPC | Shared Memory | DPC | |
|---|---|---|---|---|---|---|
| | | | | | immediate | delayed |
| IPC 데이터 복사 | $T_{ovd_fmt}$ | 임의의 값 | 임의의 값 | 임의의 값 | 0 | 임의의 값 |
| 커널로의 트랩 | $T_{th}$ | 임의의 값 | 임의의 값 | 임의의 값 | 임의의 값 | 임의의 값 |
| | $C_{kt}$ | 2 | 2 | 0 | 0 | 0 |
| | $T_{ovd_th}$ | $2T_{th}$ | $2T_{th}$ | 0 | 0 | 0 |
| 사용자와 커널공간 사이의 복사 | $T_{uk}$ | 임의의 값 | 임의의 값 | 임의의 값 | 임의의 값 | 임의의 값 |
| | $EC_{cuk}$ | 2 | 0 | 0 | 0 | 0 |
| | $T_{ovd_cuk}$ | $2T_{uk}$ | 0 | 0 | 0 | 0 |
| 접근확인 | $T_{ipc_av}$ | 임의의 값 | 임의의 값 | 임의의 값 | 임의의 값 | 임의의 값 |
| | $T_{add_av}$ | 0 | 0 | 임의의 값 >0 | 0 | 0 |
| | $T_{ovd_av}$ | $T_{ipc_av}$ | $T_{ipc_av}$ | $T_{ipc_av}+ T_{add_av}$ | $T_{ipc_av}$ | $T_{ipc_av}$ |
| 스케줄링 | $T_{ovd_sched}$ | 임의의 값 | 임의의 값 | 임의의 값 | 0 | 임의의 값 |
| 문맥교환 | $T_{cs}$ | 임의의 값 | 임의의 값 | 임의의 값 | 임의의 값 | 임의의 값 |
| | $WV_{cs}$ | 1 | 1 | 1 | 0 | $0 < WV_{cs} < 1$ |
| | $T_{ovd_cs}$ | $T_{cs}$ | $T_{cs}$ | $T_{cs}$ | 0 | $T_{cs} * WV_{cs}$ |
| RPC Stub | $T_{ovd_rpcstub}$ | 0 | 임의의 값 >0 | 0 | 0 | 0 |

- 동등한 마이크로커널 운영체계의 선택 : IPC 사건은 다른 부수적인 사건들을 필요로 하며 최종적인 결과는 관련된 모든 IPC 사건들의 처리가 끝난 후에 얻을 수가 있게 된다. 마이크로커널 운영체계는 각 시스템마다 서로 다른 서버들을 가질 수가 있다. 따라서 서로 다른 마이크로커널 운영체계에서 실행되는 IPC 방식으로서는 동일한 성능 평가의 결과를 얻을 수가 없게 된다. 본 논문에서는 동일한 마이크로커널 운영체계에서 모든 IPC 방식들을 구현하는 것으로 가정한다.
- 동등한 IPC 메시지 포맷 : IPC 방식에서 사용되는 데이터 포맷은 시스템마다 다를 수 있다. IPC 성능은 IPC 메시지 포맷의 코드 크기에 따라 종속되므로 동일한 IPC 데이터 포맷을 사용하는 것으로 가정한다.

4.3.2 매개 변수들

4.2절에서 살펴본 IPC 사건을 처리하는데 있어서 발생되는 부담의 요소들을 시간 매개변수로 설정하고 이들을 측정하여 각 IPC 방식의 상대 평가의 측정치로 계산한다.

- $T_{ovd_fmt}$ : IPC 메시지를 포맷팅하는데 소요되는 시간
- $T_{ovd_cuk}$ : 사용자와 커널사이에 IPC 데이터를 복사하는데 소요되는 시간, $T_{ovd_cuk} = T_{uk} * EC_{cuk}$.
  - $T_{uk}$ : IPC 데이터를 한 번 복사하는데 필요한 시간
  - $EC_{cuk}$ : 사용자와 커널사이에 복사의 회수
- $T_{ovd_th}$ : 커널으로의 트랩을 처리하는데 필요한 시간, $T_{ovd_th} = T_{th} * C_{kt}$
  - $T_{th}$ : 트랩을 한 번 처리하는데 필요한 시간
  - $C_{kt}$ : 커널로 트랩이 발생된 회수
- $T_{ovd_cs}$ : 문맥 교환을 처리하기 위한 시간, $T_{ovd_cs} = T_{cs} * WV_{cs}$
  - $T_{cs}$ : 문맥 교환을 처리하는데 필요한 시간
  - $WV_{cs}$ : 문맥 교환에 대한 가중치( $0 \le$ 가중치 $\le 1$ )
- $T_{ovd_av}$ : 접근 확인을 처리하기 위한 시간, $T_{ovd_av} = T_{ipc_av} + T_{add_av}$
  - $T_{ipc_av}$ : IPC 방식에 종속적인 접근 확인에 필요한 시간
  - $T_{add_av}$ : 부수적인 접근 확인에 필요한 시간( $\ge 0$)
- $T_{ovd_sched}$ : 서버를 스케줄링하는데 필요한 시간
- $T_{ovd_rpcstub}$ : RPC stub을 처리하는데 필요한 시간

위의 매개변수를 기초로 IPC 사건을 처리하는데 발생되는 부담을 표 2에서 요약하였다. 표 2의 값들중에서 "임의의 값"은 실제의 시스템에서 의미 있는 값으로 대체될 수 있음을 의미한다. 각각의 IPC 방식에서 가질 수 있는 IPC 부담을 위의 매개변수를 사용하여

식으로 나타내면 다음과 같다. 실제로 현실 세계에서 측정된 값이 아니므로 근사치로 표현한다.

o $T_{msg_pass} \cong T_{ovd_fmt} + 2T_{uk} + 2T_{th} + T_{cs} + T_{ipc_av} + T_{ovd_sched}$ (①)

o $T_{lrpc} \cong T_{ovd_fmt} + 2T_{th} + T_{cs} + T_{ipc_av} + T_{ovd_sched} + T_{ovd_rpcstub}$ (②)

o $T_{shd_mem} \cong T_{ovd_fmt} + T_{cs} + T_{ipc_av} + T_{add_av} + T_{ovd_sched}$ (③)

o $T_{dpc_imm} \cong T_{ipc_av}$ (④)

o $T_{dpc_dlyd} \cong T_{ovd_fmt} + T_{cs} * WV_{cs}(< 1) + T_{ipc_av} + T_{ovd_sched}$(⑤)

4.3.3 HP9000/730 시스템에서 상대 성능 평가

DPC IPC 방식의 성능을 평가하기 위해 다른 논문 [9]에서 측정된 RPC 테스트 결과를 이용한다. [9]에서 실험 환경은 HP9000/730 시스템을 사용하고 운영체계로서 Mach를 이용하였으며, CPU 클럭 사이클로 RPC 테스트 시간을 측정하였다. 본 논문에서 언급된 모든 IPC 방식들은 동일한 환경에서 구현된 것으로 가정하므로서 4.3.1에서 제안한 사항에 부합하는 것으로 간주한다. Mach 운영체계에서 RPC 테스트는 저하위 계층에서 메시지 전송을 사용하므로 IPC에서 발생되는 부담은 메시지 전송 IPC 방식에서 발생되는 요소들과 거의 유사하다고 할 수 있다. 따라서 RPC 테스트에서 측정된 특정 요소의 값을 이와 유사한 다른 IPC 방식에서 발생되는 요소들의 값으로 대체하기로 한다. [9]에서 보여진 RPC 테스트 결과에 의해 4.3.2절의 각 매개변수의 값을 표현하면 아래와 같다.

$T_{ovd_fmt}$ = 153 cc [3], $T_{uk}$ = 205 cc, $T_{th}$ = 350 cc, $T_{cs}$ = 178 cc

$T_{ipc_av} + T_{ovd_sched}$ = 281 cc[4], $T_{ovd_rpcstub}$ = 46 cc

식 ①~⑤에 위 매개변수의 값을 적용하면 각 IPC 방식의 부담을 시간으로 계산할 수 있는데 그 결과는 표 3에 보여진다. 표 3에서 DPC IPC 방식은 $T_{ovd_sched}$을 포함하지 않으므로 $OT_{dpc_imm}$의 결과값은 281보다 작게 된다. 또한 RPC 테스트의 결과는 $WV_{cs}$의 값을 1로 설정한 상태에서 측정된 결과이므로 $OT_{dpc_dlyd}$의 값도 765보다 작게 된다. $OT_{shd_mem}$의 경우 $T_{add_av}$를 값으로 표현할 수 없지만 대략적으로 바꿀 수는 있다.

예를 들어 공유 메모리의 접근을 보호하기위한 수단으로 세마포어(semaphore)를 이용한다고 가정하자. IPC Initiator는 공유 메모리에 쓰기 전에 먼저 P(sem) 연산을 수행하고 쓰기가 끝난후에는 V(sem)연산을 수행한다. IPC Responder는 공유 메모리부터 읽기 전에 먼저 P(sem) 연산을 수행하여야만 된다. 최악의 경우 위 세 연산이 순차적으로 발생할 수 있다. 그러한 경우에는 $T_{add_av}$는 세마포어 연산을 위해 트랩 발생과 접근 확인을 3번 수행하여야만 한다. 또한 마이크로커널은 세마포어 연산을 위해 sleep 상태에 있는 서버를 내부적으로 wake_up해야 하므로 서버 스케쥴링의 시간도 포함되어진다. 따라서 $T_{add_av}$는 $3T_{th}$와 $T_{ipc_av}$, 그리고 $T_{ovd_sched}$를 더한 값보다 클 것으로 예상된다. 표 3에 의하여 HP9000/730시스템에서 모든 IPC 방식을 구현하고 성능을 평가한다면 다음의 결과를 유추할 수 있다.[5]

DPC(immediate 모드) > DPC(delayed 모드) > LRPC > 메시지 전송 > 공유 메모리

〈표 3〉 IPC 기법들의 성능 평가결과
〈Table 3〉 Results of performance evaluation for IPC Schemes

| IPC 부담 시간 (OT: Overhead Time) | Clock Cycles (HP PA-RISC 1.1 Processor) |
|---|---|
| $OT_{msg_pass}$ | ≃ 1722 |
| $OT_{ipc}$ | ≃ 1358 |
| $OT_{shd_mem}$ | ≃ 612 + $T_{add_av}$ |
| $OT_{dpc_imm}$ | < 281 |
| $OT_{dpc_dyld}$ | < 612 |

## 5. 결 론

다중개성 운영체계는 응용프로그램의 요구에 보다 쉽게 대처할 수 있도록 마이크로커널 구조의 운영체계로서 구성되어진다. 마이크로커널 구조의 운영체계는 서로 다른 주소 공간을 가지는 독립적인 서버와 커널로 구분되어지며, 사용자와 서버 혹은 서버 사이의 통신을 위해서 사용되어지는 메시지 전송 방식의 비용 부담때문에 전체적인 시스템 성능의 저하를 가져온다. 본 논문에서는 마이크로커널 방식의 시스템에서 성능 향상을 위해 새로운 IPC 방식인 DPC IPC 방식과 운영체계에 관련된 여러 시스템 설계 사항을 새로이 고

3) IPC 메시지 포맷은 32 words이다.
4) 테스트 결과로부터 매개변수를 분리할 수 없었다.
5) 여기서 >는 성능이 우수함을 의미한다.

려하였다. 특히 DPC IPC 방식에 대해 다음과 같은 결과를 얻을 수 있었다.

- 메시지 포맷팅을 제거할 수 있다.
- 사용자와 커널 사이의 복사 연산을 완전히 제거한다.
- 송/수신자 사이의 문맥 교환의 정도를 훨씬 감소시킬 수 있게 된다.
- 커널으로의 트랩은 불필요하다.
- 서버 스케줄링은 단순한 연산으로 처리할 수 있으며 그에 대한 부담도 훨씬 작게 유지할 수 있다.

성능 평가의 결과로서 마이크로커널 시스템에서 DPC IPC 방식을 채택할 경우에 다른 통신 방식을 채택한 시스템보다도 훨씬 좋은 성능을 가질 수 있음을 발견하였다.

## 참 고 문 헌


[1] Andrew S. Tanenbaum, "Modern Operating System", Prentice Hall, international edition, 1992.

[2] ATT, "Programmer's Guide: ANSI C and Programming Support Tools", chapter 13, Prentice-Hall, 1990.

[3] Brian N. Bershad and Thomas E. Anderson, "Lightweight Remote Procedure Call", ACM Transactions on Computer Systems, 8(1):18-55, 1990.

[4] Daniel P. Julin, Jonathan J. Chew, and J. Mark Stevenson, "Generalized Emulation Services for Mach 3.0", Proceedings of the Second USENIX Mach Symposium, 1991.

[5] David Keppel, Susan J. Eggers, and Robert R. Henry, "A Case for Runtime Code Generation", Technical Report 91-11-04, University of Washington, 1991.

[6] Eric J. Koldinger, Jffrey S. Chase, and Susan J. Eggers, "Architectural Support for Single-Address Space Systems", Technical Report 92-03-10, University of Washington, 1992.

[7] Gintaras R. Gircys, "Understanding and Using COFF", O'Reilly, 1988.

[8] James Kempf and Peter B. Kessler, "Cross-Address Space Dynamic Linking", Technical Report TR-92-2, Sun Microsystems Laboratories, 1992.

[9] Jay Lepreau, Mike Hibler, Bryan Ford, and Jeffrey Law. "In-Kernel Servers on Mach 3.0", Technical Report, University of Utah, 1993.

[10] Jay Lepreau, Mike Hibler, Bryan Ford, Jeffrey Law, and Douglas Orr, "In-Kernel Servers on Mach 3.0: Implementation and Performance," Proceedings of the USENIX MACH III Symposium, April 1993.

[11] Jeffrey S. Chase, Henry M. Levy, Michael J. Feeley, and Edward D. Lazowska, "Sharing and Protection in a Single-Address-Space Operating System", ACM Transactions on Computer Systems, 12(4):271-305, 1994.

[12] Jeffrey S. Chase, Henry M. Levy, Edward D. Lazowska, and Michael Baker Harvey, "How to User a 64-Bit Virtual Address Space", Technical Report 92-03-02, University of Washington, 1992.

[13] Jeffrey S. Chase, Henry M. Levy, Edward D. Lazowska, and Michael Baker Harvey, "Lightweight Shared Objects in a 64-Bit Operating System", Technical Report 92-03-09, University of Washington, 1992.

[14] Jeffrey S. Chase, Rene W. Schmidt, and Henry M. Levy, "Shared Memory Support for Object-based RPC", In the Univ. of Washington, February, 1995

[15] Joseph Boykin, David Kirschen, Alan Langerman, and Susan LoVerso, "Programming Under Mach", Addison-Wesley, 1993.

[16] Maurice J. Bach, "The Design of the UNIX Operating System", Prentice Hall, international edition, 1986.

[17] Michael J. Feeley, Jeffrey S. Chase, and Edward D. Lazowska, "User-Level Threads and Interprocess Communication", Technical Report 93-02-03, University of Washington, 1993.

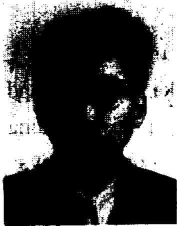
[18] Open Software Foundation, "Mach 3 Kernel Principles", July 1992.

[19] Sammuel J. Leffler, Marshall Kirk McKuick, Michael J. Karels, and John S. Quaterman. "The Design and Implementation of the 4.3 BSD UNIX

Operating System", Addison-Wesley, 1988.

[20] Sun Microsystems, "Programming Utilities and Libraries", 1988.

[21] Timothy James Wilkinson, "Implementing Fault Tolerance in a 64-bit Distributed Operating System", PhD thesis, City University, London, England, July 1993.



**조 시 훈**

1992년 경북대학교 컴퓨터과학과 (학사)

1994년 한국과학기술원 정보통신공학과(석사)

1994년~현재 한국과학기술원 전산학과 박사과정

관심분야 : 운영체계, 컴퓨터 시스템, 병렬처리, 성능 평가 등임

**방 남 석**

1986년 한국항공대학 항공통신공학과(학사)

1996년 한국과학기술원 정보통신공학과(석사)

1986년~현재 삼성전자(주) 영상응용그룹 선임연구원

관심분야 : 다중처리기구조, 운영체계, 멀티미디어 응용 (VOD, Video Conferencing, Digital broadcasting) 등임

**이 준 원**

1983년 서울대학교 계산통계학과 (학사)

1990년 Georgia Tech. 전산학과 (석사)

1991년 Georgia Tech. 전산학과 (박사)

1983년~1986년 (주)유공 근무

1991년~1992년 IBM 근무

1992년~현재 한국과학기술원 전산학과 조교수

관심분야 : 다중처리기구조, 운영체계, 컴퓨터 시스템, 병렬처리 등임