# Linux 운영체제에서 Shared Memory 성능 개선 방안 연구

장승주[1], 최은석[1], 강동욱[2], 이광용[2], 김동한[2], 김재명[2]
[1]동의대학교 컴퓨터공학과
sjjang@deu.ac.kr
[2]한국전자통신연구원(ETRI)
dkang@etri.re.kr

# A Study of Performance Enhancement for the Shared Memory in the Linux O.S

Seung-Ju Jang[1], Eun-Seok Choi[1], Dong-Uk Kang[2], Gwang-Yong Lee[2], Dong-Han Kim[2], Jae-Myeong Kim[2]
[1]Dept. of Computer Engineering, Dong-Eui University
[2]Electronics and Telecommunications Research Institute(ETRI)

**요 약**

본 논문은 대부분의 Linux 운영체제에서 지원해 주는 System V의 IPC 중 하나인 Shared Memory의 성능을 개선하는 방안을 연구한다. Linux에서 사용되는 Shared Memory는 동일한 메모리 영역에 여러 개의 프로세스가 접근할 수 있도록 해 주는 기술이다. 본 논문에서는 Shared Memory의 큰 두 갈래 중 커널 단계에서 처리 되는 SVR 형식의 Shared Memory를 다룬다. 본 논문에서는 리눅스 운영체제의 공유 메모리 성능 개선 방안을 제안한다. 본 논문에서 제안하는 공유 메모리 성능 개선 방안은 듀얼 코어를 활용하여 기존의 단일 처리기 시스템에서보다 성능을 향상시킬 수 있도록 한다. 공유 메모리를 이용한 프로세스의 동작이 별개의 CPU에서 동작되도록 함으로써 성능 향상을 꾀한다.

## 1. 서 론

Linux에서 사용되는 Shared Memory(공유메모리)는 동일한 메모리 영역에 여러 개의 프로세스가 접근할 수 있도록 해 주는 매력적인 기술이다. 일반적으로 SVR IPC(System V Release InterProcess Communication)라 불리는 공유메모리, 세마포어, 메시지 큐의 세 가지는 System V 유닉스에서 구현된 기술이다.

본 논문에서는 SVR IPC에서 사용되는 기술 중 공유메모리(Shared Memory)에 대한 성능 개선을 연구한다. 복잡한 프로그래밍 환경에서 다수의 프로세스들은 서로 협력하기 위하여 서로 통신하고 자원과 정보를 공유한다. 커널에서는 이것이 가능한 방법을 제공하는 데, 이를 프로세스간 통신(Inter-Process Communication) 또는 IPC라 부른다. 프로세스간 통신을 하는 목적은 데이터 전송, 데이터 공유, 사건 전송, 자원 공유, 프로세스 제어를 하기 위함이다. 이를 위하여 전통적인 Unix에서는 시그널(Signal), 파이프(Pipe), 프로세스 추적(Process Tracing)의 기능을 사용하였다. 그러나 많은 사용자 프로그램들에서는 이것만으로 IPC를 충족할 수가 없었으며, 이것은 SystemV 릴리즈 버전에서 메시지 큐, 세마포어, 공유 메모리 기법을 사용하게 된다.

Shared Memory의 구현에 있어서는 시스템의 가상 메모리(Virtual Memory) 구조에 크게 의존하며, 데이터의 복사나 시스템 요청을 사용하지 않고 많은 양의 데이터를 공유하는 매우 빠르고 융통성 있는 기법을 제공한다. 그러나 이 기법은 동기화 기법을 제공하지 않는다는 단점이 있다. 두 개의 프로세가 있다고 가정하면, 공유 메모리 영역의 내용을 동시에 변경하려고 할 경우, 커널은 이를 순차적으로 처리를 해야 하나, 그렇게 하지 않으면, 쓰인 데이터는 엉키게 된다. 그래서 공유 메모리를 사용하는 프로세스들은 프로세스들 간에 동기화를 하는 프로토콜을 고안해야 하며, 이것은 공유 메모리 성능에 영향을 주는 요소로 적용할 수 있다.

본 논문은 2장에서 Shared Memory에 관하여 설명하고, 3장에서 Shared Memory의 성능 개선 방향에 대해 설명하고, 4장에서 Shared Memory의 성능을 측정하는 프로그램을 설명한다. 5장에서 Shared Memory의 성능을 측정하고, 6장에서 결론을 맺는다.

## 2. 관련연구

유닉스의 두 갈래 중 하나인 SVR은 초기 상용 유닉스 버전들이 해당한다고 보면 되며, 다른 한 갈래인 BSD는 버클리 대학에서 독자적으로 구현한 것이다. 초기 BSD는 대학 내에 설치된 상용 유닉스 위에 학생들이 필요한 소프트웨어를 만들며 시작된, 일종의 Free Software 모음이었으나 그 덩치가 커지면서 OS 단계까지 구현하게 되었다.

결국 상용 유닉스 업체의 법적 소송으로 인해 완전히 소스를 분리하여 새로운 유닉스 버전을 만들게 되었다.

BSD와 SVR은 서로 다른 길을 걷게 되었지만 나중엔 다시 합쳐져서 현재의 상용 유닉스들은 대부분 통합된 패키지를 제공하고 있다.

그 예로, TCP/IP 관련은 BSD에서 먼저 구현되었으며, SVR은 IPC API들을 구현하였다. 현재 BSD의 후계자인 FreeBSD 등의 유닉스에서는 SVR IPC를 직접적으로 지원하지는 않고 있다.

IPC의 Shared Memory는 shmget( )으로 시작하는 SVR 형식의 Shared Memory와 shm_open()으로 시작하는 POSIX 형식의 Shared Memory로 크게 나뉜다.
두 가지 Shared Memory의 큰 차이점은 SVR 버전은 커널에서 구현되었고, POSIX 버전은 라이브러리로 구현되었다. POSIX 버전의 Shared Memory는 일반 시스템 콜을 활용한 라이브러리 이다.

널리 사용되는 SVR 버전은 대부분의 유닉스 커널과 리눅스에서 지원하며 커널에 포함되어 있기 때문에 커널 파라미터를 조정하여야 제한 사항을 바꿀 수 있다. API를 통해 Shared Memory를 할당 받으면 해당되는 메모리 영역의 주소를 받을 수 있다. 메모리 영역의 구분은 key와 id를 통해 다른 프로세스가 접근 할 수 있으며 메모리 영역별로 euid라든가 permission 같은 메타 정보 역시 따로 커널에서 관리하게 된다.

반면 POSIX 버전은 일종의 메모리 디스크 방식으로 디렉터리에 마운트 한 후, 거기에 파일을 만드는 식으로 접근을 한다. 파일은 기본적으로 여러 프로세스에서 접근할 수 있는 개체이고 이 파일을 메모리 상에 만드는 것이므로 우회적으로 Shared Memory의 구현이 가능한 것이다. 그리고 메모리에 만든 이 파일을 mmap( )으로 매핑하면 실제 메모리를 쓰듯이 쓰게 된다. 파일이란 개체에 원래 permission과 euid등의 정보가 존재하므로 SVR 버전처럼 이에 대한 별도의 관리도 필요 없으며 커널에서는 파일로 인식할 뿐이다.
Shared Memory에 대해 SVR 버전은 '특수한 메모리 영역'으로 지정하여 사용자에게 사용하게 해 준다. POSIX 버전은 일반 메모리를 파일시스템처럼 써서 공유가 가능하게 한 후 다시 mmap( )으로 하여 메모리처럼 쓰게 해준다. mmap( )은 실제 파일이라도 파일 오프셋에 가상 주소를 할당해 메모리처럼 쓸 수 있게 해준다.
하지만 POSIX Shared Memory는 느리다. SVR처럼 커널 단계에서 커널 모듈을 거쳐 직접 메모리에 접근하는 것이 아닌, 라이브러리 단계에서 mmap( )과 파일 시스템 API를 거쳐서 간접적으로 메모리에 접근하기에 느리다.

POSIX Shared Memory가 단점만 있는 것은 아니다. 장점은 파일 형태로 구현되어 있기 때문에 사용 하는 만큼만 메모리가 늘어난다는 점이다. SVR의 경우 미리 쓸 만큼의 메모리를 할당 받고 사용하며, 할당량 보다 크게는 사용 할 수 없다.

### 3. Shared Memory 성능 개선 방안

Shared Memory는 물리적인 메모리의 특정 영역에 대하여 프로세스들에 의해 공유되어지는 공통의 영역을 구성한다. Shared Memory는 공통의 영역을 프로세스들이 사용할 수 있게 해준다.

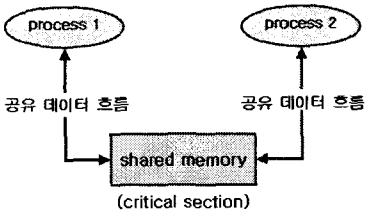


그림 1 Shared Memory 구조

그림 1은 Shared Memory의 구조를 나타낸다. 본 논문에서 연구하는 Shared Memory는 물리적인 메모리의 특정 영역에 대하여 프로세스들에 의해 공유되어지는 그림 1의 Critical Section(영역)이라 할 수 있다. 프로세스들은 이 Shared Memory 영역을 자신의 가상 메모리 영역에 부착(Attach)하여 사용한다. 영역은 데이터 읽기/쓰기의 시스템 호출을 사용하지 않고 다른 방법으로 접근하기 때문에 프로세스 간 데이터를 공유하는 기법 중 가장 빠르다 할 수 있다. 예를 들어, 하나의 프로세스가 공유 메모리의 영역에 기록(Write)을 하게 된다면, 이 내용은 이 영역을 공유하는 모든 프로세스들에 의해 바로 보여 지게 되어 아주 빠르게 데이터를 공유하게 된다.

Shared Memory의 성능 개선 방법으로 CPU를 여러 개를 사용하는 방법이 있다. 이는 쉬운 방법이긴 하나 비용적인 측면에서는 좋은 방법이라고 할 수는 없다.
CPU(Central Processing Unit)는 컴퓨터 시스템에서 차지하는 비용이 절반 가까이 차지할 정도로 고가의 부품이다.
CPU등의 하드웨어 장비를 Upgrade 하는 것으로 Shared Memory의 성능을 올리는 것은 누구나 생각할 수 있는 방법인 것이다.

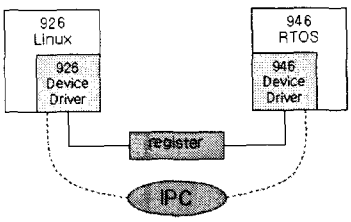


그림 2 2CPU에서의 IPC 구조

그림 2는 2 CPU에서의 IPC(본 논문에서는 Shared Memory)의 구조이다. 서로 다른 CPU에 서로 다른 운영체제가 IPC 메카니즘 중 Shared Memory 기법으로 서로의 데이터를 공유하는 모습이다.
그림 2와 같이 서로 다른 CPU를 사용할 경우에 각각의 CPU에서 Shared Memory를 통하여 데이터를 공유하거나 정보를 주고받을 수 있다. 또한 서로 다른 CPU에서 동시에 다른 프로세스가 수행될 수 있다. 프로세스들의 수행양이 많을 경우 단일 프로세서에서는 프로

세스 수행에 시간이 많이 소용된다. 이는 단일 프로세서는 프로세스들을 순차적으로 처리하기 때문이다. 한 프로세스가 CPU 사용의 최대 시간(time slice)을 넘을 정도로 긴 수행을 해야 하는 경우라 가정한다. 단일 프로세서는 CPU 사용의 최대 시간을 사용한 프로세스1에 대해서 CPU 사용을 중지 시키고 프로세스1은 Sleep 상태로 들어서고, 다음 사용자인 프로세스2가 CPU를 할당 받는다. 이럴 경우 프로세스 수행 시간의 거의 2배 가까운 시간이 소요되어서야 프로세스의 수행이 종료될 수 있다. 하지만 그림 2와 같이 두 개의 CPU를 사용하게 된다면, 혹은 두 개의 CPU를 사용하는 것과 같은 듀얼 코어 프로세서를 사용하게 된다면 두 개의 프로세스를 동시에 수행할 수 있어 성능 향상에 도움이 된다. 본 논문에서 제안하는 공유 메모리 성능 개선 방안은 듀얼 코어를 활용하여 기존의 단일 처리기 시스템에서보다 성능을 향상시킬 수 있도록 한다. 공유 메모리를 이용한 프로세스의 동작이 별개의 CPU에서 동작되도록 함으로써 성능 향상을 꾀한다.

본 논문에서는 듀얼 코어가 단일 프로세스보다 Shared Memory에서도 성능이 뛰어 남을 보이기 위해 우선적으로 듀얼 코어에서의 Shared Memory 성능을 5장에서 측정한다.

## 4. Shared Memory 성능 측정 프로그램

Shared Memory를 사용한 예제 프로그램들은 인터넷 상이나, Shared Memory를 설명하는 책에서 쉽게 접할 수 있다.

본 논문에서도 Shared Memory의 성능을 측정하기 위한 프로그램으로 인터넷 상에서 Shared Memory를 설명하는 예제 소스를 사용하였다.

성능을 측정하기 위하여 예제 소스에 시간을 측정 할 수 있는 기능을 추가 해 주어야 한다.

```
#include <sys/ipc.h>
#include <sys/shm.h>
#include <string.h>
#include <unistd.h>

int main()
{
        int shmid;
        int pid;
        int *cal_num;                                   - ①
        void *shared_memory = (void *)0;
        shmid = shmget((key_t)1234, sizeof(int),
                        0666|IPC_CREAT);

        if (shmid == -1)
        {
                perror("shmget failed : ");
                exit(0);
        }

        shared_memory = shmat(shmid, (void *)0, 0);

        if (shared_memory == (void *)-1)
        {
                perror("shmat failed : ");
                exit(0);
        }

        cal_num = (int *)shared_memory;
        pid = fork();

        if (pid == 0)                                   - ②
        {
                shmid = shmget((key_t)1234, sizeof(int), 0);
                if (shmid == -1)
                {
                        perror("shmget failed : ");
                        exit(0);
                }
                shared_memory = shmat(shmid, (void *)0,
                                0666|IPC_CREAT);
                if (shared_memory == (void *)-1)
                {
                        perror("shmat failed : ");
                        exit(0);
                }
                cal_num = (int *)shared_memory;
                *cal_num = 1;
                while(1)
                {
                        *cal_num = *cal_num + 1;
                        printf("child %d\n", *cal_num);
                        sleep(1);
                }
        }

        else if(pid > 0)                                - ③
        {
                while(1)
                {
                        sleep(1);
                        printf("%d\n", *cal_num);
                }
        }
}
```

그림 3 Shared Memory 예제 프로그램

그림 3은 본 논문에서 Shared Memory의 성능을 측정하기 위해 사용한 예제 프로그램의 원래 소스 코드이다. 그림 3의 코드는 fork( ) 시스템 콜을 사용하여 부모 프로세스(그림 3의 ③)와 자식 프로세스(그림 3의 ②)로 나누어 두 개의 프로세스가 동일한 변수인 "cal_num" (그림 3의 ①)을 쓰고 읽는 과정을 출력하는 프로그램이다.

그림 3의 ②는 자식 프로세스가 동작하는 부분을 나타내고 있다. 자식 프로세스는 Shared Memory를 고유 Key 값인 '1234'로 영역에 접근 할 수 있는 id를 shmget( )함수로부터 부여 받는다. 부여 받은 고유 id로 shmat( )함수를 이용하여 Shared Memory에 접근을 한다. 이 때 Shared Memory 영역이 이미 있다면 기존의 영역에 접근을 하고, 만약 Shared Memory 영역이 없다면 Shared Memory 영역을 생성한 후 접근한다.

그림 3의 ③인 부모 프로세스는 fork( )함수가 사용되

기 이전에 Shared Memory 영역을 확보하였고, 사용할 수 있게 되어 있어 shmget( )함수나, shmat( )함수를 사용하지 않고 바로 Shared Memory 영역에 접근 할 수 있다.

```
#include <sys/ipc.h>
#include <sys/shm.h>
#include <string.h>
#include <unistd.h>

#include <time.h> /* 성능 Test를 위한 시간관련 함수 헤더 */
#define MAX_NUM 5000

int main()
{
        int counter; /* 유한한 동작을 위한 카운터 */
        time_t start, end; /* 성능 Test를 위한 변수 */

        int shmid;
        int pid;
        int *cal_num;
        void *shared_memory = (void *)0;

        start = time(0); /* 시작시간을 0으로 초기화 */
        shmid = shmget((key_t)1234, sizeof(int),
                0666|IPC_CREAT);
        if (shmid == -1)
        {
                perror("shmget failed : ");
                exit(0);
        }

        shared_memory = shmat(shmid, (void *)0, 0);

        if (shared_memory == (void *)-1)
        {
                perror("shmat failed : ");
                exit(0);
        }

        cal_num = (int *)shared_memory;
        pid = fork();

①      if (pid == 0)
        {
                shmid = shmget((key_t)1234, sizeof(int), 0);
                if (shmid == -1)
                {
                        perror("shmget failed : ");
                        exit(0);
                }
                shared_memory = shmat(shmid,(void *)0,
                                0666|IPC_CREAT);
                if (shared_memory == (void *)-1)
                {
                        perror("shmat failed : ");
                        exit(0);
                }
                cal_num = (int *)shared_memory;
                *cal_num = 1;
                counter = 1;
                while(counter < MAX_NUM)
                {
                        counter = counter + 1;
                        *cal_num = *cal_num + 1;
                        printf("child %d\n",
                                *cal_num);
                        sleep(1);
                }
        }

②      else if(pid > 0)
        {
                counter = 1;
                while(counter < MAX_NUM)
                {
                        counter = counter + 1;
                        sleep(1);
                        printf("%d\n", *cal_num);
                }
                end = time(0); /* 끝 시간을 초기화 */
                printf("Execution time = %.3f sec.\n",
                        difftime(end, start));
        }
}
```

그림 4 Shared Memory 성능 측정 프로그램

그림 4는 그림 3의 소스 코드에 시간을 측정할 수 있는 부분을 추가하여 완성한 성능 측정 프로그램이다. 시간을 측정하여 성능이 좋고 나쁨으로 나뉘기 때문에 시간 관련 함수를 사용 할 수 있게 해주는 'time.h'를 헤더에 추가하였다. 그리고 그림 3의 소스 코드의 경우 무한 루프를 수행하게 프로그램 되어 있다. 그 이유는 그림 3의 ②와 ③에서 사용된 "while(1)" loop문이 항상 참을 나타내기 때문이다. 따라서 while loop문의 loop 횟수를 조절 하여 유한한 시간동안 동작하게 설계한다.

시간측정 방법은 프로그램의 도입부에서 time( ) 함수를 사용하여 시작 시간을 저장하고, 프로그램이 종료되는 부분에서 끝 시간을 저장한다. 시작 시간과 끝 시간의 차이를 difftime( )함수로 얻는다.

그림 4의 소스 코드를 컴파일 하여 5장의 시스템에서 성능을 측정한다.

## 5. Shared Memory 성능 측정

본 논문에서는 듀얼 코어 시스템에서 Shared Memory의 성능을 측정 하였다. 본 논문의 4장에서 보여준 예제 프로그램에 수행 시간 계산 능력을 추가 한 프로그램으로 Shared Memory의 성능을 측정한다.

본 논문에서는 Linux 기반의 Kernel 2.4.20-8 버전의 Shared Memory 기능으로 테스트 하였다. Kernel 2.4.20-8 버전은 Red Hat 9.0에서 제공하는 커널 버전이다. Kernel 2.4.20-8 버전은 본 논문의 3장에서 언급한 듀얼 코어 시스템을 사용하기 위한 smp(Symmetric Multi-Processing)기능을 지원한다.

smp 기능은 여러 CPU를 사용하는 시스템에 적합한 환경을 제공한다. Unix 기반인 Linux 운영체제는 그 특징에서 알 수 있듯이 소형의 임베디드 기기부터 대형의

슈퍼컴퓨터까지 그 쓰임이 다양한 것이 이러한 smp 기능 등을 지원해 주기 때문이다.

표 1 성능 측정에 사용된 시스템 환경

| | | |
|---|---|---|
| CPU | Intel® Core™ 2 CPU 6400 | 2.13GHz, 2.13GHz |
| RAM | 1 GB | DDR2 512MB × 2 |
| HDD (총 할당 공간) | 30 GB | SATA2 방식 |
| linux Version | Red Hat 9.0 (2.4.20-8smp) | kernel 2.4.20-8 |
| gcc Version | 3.2.2 | Red Hat linux 3.2.2-5 |

표 1은 Shared Memory의 성능을 측정한 시스템의 환경을 나타낸다. 듀얼 코어로 사용된 시스템은 Red Hat Linux 9.0을 기반으로 하고 있다.

듀얼 코어를 지원하기 위한 smp(Symmetric Multi-Processing) 기능을 갖춘 커널을 사용하였다. 듀얼 코어는 물리적으로 두 개의 프로세서 코어를 하나로 통합, 집적화한 것으로 겉보기에는 하나의 중앙 처리 장치(CPU)로 보이지만, 실제로는 두 개에 해당하는 CPU를 단 것처럼 강력한 연산 능력을 보이는 것으로 향후 프로세서 로드맵에 있어 기반 기술로 기대하고 있다. 듀얼 코어 방식은 한국 IBM사의 파워5, 한국 선 마이크로시스템스사의 스팍4, 한국 휴렛팩커드(HP)사의 PA8800이 채택하고 있다. 듀얼 코어 프로세서는 물리적으로는 두 개이지만 용도에 따라서는 하나의 프로세서로 볼 수도 있어 프로세서 당으로 적용하는 소프트웨어 라이선스 정책에 혼선으로 줄 수도 있다.

표 2 듀얼 코어에서 Shared Memory 성능 측정 결과

| 번호 | 루프 수 | 소요 시간(초) | 비 고 |
|---|---|---|---|
| 1 | 10 | 9 | 9초 |
| 2 | 300 | 302 | 5분 2초 |
| 3 | 600 | 605 | 10분 5초 |
| 4 | 1600 | 1615 | 26분 55초 |
| 5 | 2000 | 2019 | 33분 39초 |
| 6 | 3000 | 3029 | 50분 29초 |
| 7 | 4000 | 4039 | 1시간 7분 9초 |
| 8 | 5000 | 5049 | 1시간 24분 9초 |
| 9 | 333000 | 33632 | 9시간 20분 32초 |

표 2는 듀얼 코어에서 Shared Memory의 성능을 측정한 결과를 표로 나타내었다.

표 2에서는 그림 4의 코드에서 정의된 MAX_NUM 값에 따라 루프 수가 결정된다. 표 2의 소요 시간 결과는 9번의 루프 수 333000 을 제외하고는 모두 3번 이상 테스트 하여 그 결과의 평균값을 기록하였다. 공정한 테스트 결과를 얻기 위하여 테스트 시에는 테스트만 수행하였다.

표 2의 9번인 루프 수 333000의 경우 9시간이 넘는 수행 시간을 보인다. 이는 표 2의 6번인 루프 수 3000을 수행하려 그림 4의 소스 코드를 수정하다 숫자 3이 두 번 더 들어간 상황이라 테스트를 한번만 수행하였다.

## 6. 결론

최근 컴퓨터 하드웨어의 눈부신 발전으로 인하여 고급 부품들의 가격이 점점 내려가고 있다. 컴퓨터 시스템의 두뇌라 할 수 있는 CPU의 가격이 단일 코어와 듀얼 코어 간의 차이가 점점 좁혀 지고 있다. 또한 듀얼 코어를 넘어선 쿼드 코어 시스템도 등장하고 있다.

현재는 서버 시스템만이 여러 CPU를 사용하는 시대는 지났다. 일반인들이 사용하는 범용 PC에서도 여러 개의 CPU까지는 아니더라도 여러 개의 코어를 사용하고 있어 멀티프로세싱의 시대를 열고 있다.

이러한 시점에서 여러 코어에서 여러 프로세스가 동작하고, 그 프로세스들 간의 통신이 중요하게 된 시점에서 본 논문에서 제안하고자 하는 IPC 메카니즘 중 Shared Memory의 성능 개선은 중요한 과제이다. 본 논문에서 제안하는 공유 메모리 성능 개선 방안은 듀얼 코어를 활용하여 기존의 단일 처리기 시스템에서보다 성능을 향상시킬 수 있도록 한다. 공유 메모리를 이용한 프로세스의 동작이 별개의 CPU에서 동작되도록 함으로써 성능 향상을 꾀한다. 본 논문은 대부분의 유닉스 시스템에서 사용되는 IPC 메카니즘 중 Shared memory의 성능을 개선하기 위해 성능을 측정 해 보았다.

성능 측정에 사용된 시스템은 Intel Core2 CPU를 갖추고 있으며, 메모리는 DDR2방식의 1GB 용량의 RAM을 갖고 있다. 설치된 운영체제는 Kernel 버전 2.4.20-8로 Red Hat 9.0의 기본 커널을 사용한다.

본 논문에서 연구하는 Shared Memory의 성능 향상을 위한 기본 자료 확보 방안으로 두 개의 코어를 가지고 있는 듀얼 코어 CPU에서 성능 측정을 하였다. 듀얼 코어에서 Shared Memory의 성능을 측정하여 결과를 얻었다.

## 참고문헌

[1] http://eminency.egloos.com/
[2] http://faqs.org/docs/kernel_2_4/
[3] Uresh Vahalia, "UNIX Internals", 홍릉과학출판사, 2001
[4] 한동훈, "시스템V IPC - 공유메모리", 1997
[5] 김용준, "공유메모리를 이용한 채팅방의 원리"
[6] 한동훈, "공유메모리 vs 세마포어를 이용한 chat program", 1997
[7] Sean Walberg, "Share application data with UNIX System V IPC mechanisms", IBM, 2007
[8] 박찬모, "분산 공유 메모리 시스템 설계에 관한 연

구", 조선대학교 동력자원연구소 동력자원연구소지 19권 1호 p.129-143, 1997.05
[9] 이병관, "분산 공유 메모리에서 일관성 제어 프로토콜", 관동대학교 부설 산업기술개발연구소 산업기술논문집 12호 p.69-78, 1997.10
[10]이상권 외 3명, "KDSM(DAIST Distributed Shared Memory) 시스템의 설계 및 구현", 한국정보과학회 논문지:시스템및이론 29호 p.257-264, 2002
[11]박기홍, "Shared Memory를 갖는 멀티프로세서 시스템 구현에 관한 연구", 군산대학 자원과학연구소 3호 p97-103, 1988
[12]양제현, "Scalable Synchronization in Shared Memory Multiprocessing System", University of Marland, 1994.

**본 연구는 정보통신부 및 정보통신연구진흥원의 IT신성장동력핵심기술개발사업의 일환으로 수행하였음. [2006-S-038-02, 모바일 컨버전스 컴퓨팅을 위한 단말적응형 임베디드 운영체제 기술]**