대형 실시간 시스템의 응용 소프트웨어 개발을 위한 능동적 메모리 개체로서의 공유 라이브러리

정부금*, 차영준, 김형환, 임동선

한국전자통신연구원 실시간 OS 팀

(Tel) 042-860-5315, (Fax) 042-860-5410, bgjung@etri.re.kr

A Shared Library as an Active Memory Object for Application Software Development of Large Scale Real-time Systems

Boo-Geum Jung*, Young-Jun Cha, Hyung-Hwan Kim, Dong-Sun Lim

Real-Time OS Team, Electronics and Telecommunications Research Institute (ETRI)

(Tel) 042-860-5315, (Fax) 042-860-5410, bgjung@etri.re.kr

Abstract

In this paper, we present a novel approach named a shared library as an active memory object for application software development of large-scale real-time systems. Unlike the general passive shared memory, shared library proposed in this paper can be activated as an execution object. Moreover this is not tightly coupled with application programs unlike the normal libraries. To implement this mechanism, operating system makes the shared memory as an active object and shared library realizes the indirect call structure. This mechanism enhanced the utilization of main memory and communication performance. And this is successfully applied to the HANbit ACE ATM switching system and the TDX-10 switching system.

1. 서론

대형 실시간 시스템에 적용되는 응용 소프트웨어는 많은 사람이 참여하여 다수 개의 기능 블록으로 분할하고 각각 독립적으로 병행 개발을 하여 전체를 통합하는 대형 소프트웨어 패키지의 형태로 수행이 된다. 이렇게 개발된 각 기능 블록들은 상호 간에 긴밀한 연관 관계를 가지면서 동작되게 되는데 이때 여러 블록들에서 공통으로 사용하는 펑션들이 있을 수 있으며 공통으로 관리해야 하는 데이터들이 있게 된다. 또한 서로 독립된 블록들 간에 정보를 전달해야 한다.

이를 위한 기존의 대응 방법은 우선, 공통 펑션은 라이브러리 형태로 사용자 블록에 밀 결합 시킬 수 있으며 공통 데이터는 공유 메모리 기능을 사용하여 관리할 수 있다. 또한 블록들 간의 정보 전달은 IPC (Inter-Process Communication) 기능을 통하여 메시지를 주고 받을 수 있다[1].

이러한 기존의 방법들을 그대로 실시간 시스템에 적용시킬 경우 다음과 같은 문제점들이 있다. 우선 밀결합 라이브러리의 양이 많아질 경우 재사용이 가능한 개체의 중복 사용으로 실시간 시스템에서 거반이 되는 자원인 메인 메모리 사용의 효율성이 저하된다. 또한 공유 메모리를 사용하여 공통 데이터를 관리할 경우 메시지 전달 외에 또 다른 인터페이스를 사용하여야 하므로 관리의 다원화로 프로그램의 복잡도가 높아 진다. 또한 IPC 만을 사용하여 정보를 전달할 경우 프로그램은 단순해지나 성능 향상에 장애가 될 수 있다.

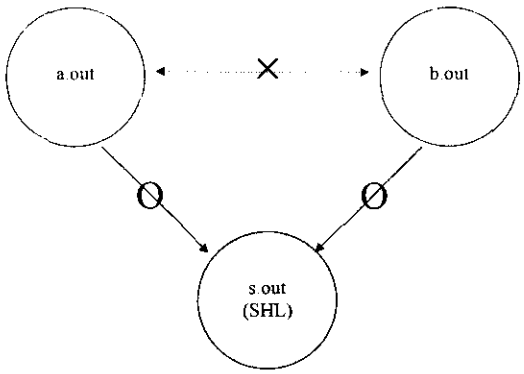
이러한 문제점들을 해결하기 위한 방법으로 본 논문에서는 실시간 시스템의 메모리 사용의 효율성을 높이며 공통 소프트웨어 모듈을 부품화 하여 이의 재사용성을 높이면서 사용자에게 편리한 인터페이스를 제공하고 실시간 시스템의 성능상의 제약 조건을 만족하는 새로운 방법으로 능동적 공유 메모리 개체로 동작할 수 있는 공유 라이브러리를 제안한다.

제안된 능동적 공유 라이브러리는 일반 라이브러리와 달리 이를 사용하는 블록과 전혀 결합되어 있지 않으면서도 실행 주체가 되어 서비스를 제공할 수 있는 능동성을 가지며, 공유 메모리의 성격을 가지므로 IPC를 통한 메시지 전달의 부담이 없는 장점을 가지므로 대형 실시간 시스템의 응용 소프트웨어의 구조적 및 수행상의 효율을 높일 수 있는 기법이다.

2. 공유 라이브러리의 개념 및 특성

다수 개의 사용자 블록과 그 내부에서 다시 다수개의 프로세스가 생성되어 동작되는 멀티 프로그래밍 및 멀티 프로세싱 환경[2]에서는 각 프로세스 별로 독립된 주소 공간을 갖게 된다. 예를 들어 사용자 블록들을 a.out 에서 z.out 이라고 이름하고 공유 라이브러리 블록

을 s.out 이라고 이름한다면 s.out 을 제외한 a.out 에서 z.out 의 모든 블록들은 자신의 실행 환경을 위하여 각각 나름대로 고유한 주소 공간을 갖는다. 즉, a.out 의 특정 주소와 b.out 의 특정 주소가 동일한 값이라고 하더라도 실제 시스템 내의 물리적 번지는 전혀 다른 위치를 가리키게 되어 서로 독립적으로 수행되는 환경을 갖게 되고 자신 외의 다른 블록들은 접근할 수가 없다. 일반적인 사용자 블록이 이러한 특성을 갖는 반면 공유 라이브러리(SHL: SHared Library)는 모든 사용자 블록들에서 그 펑션들 및 데이터를 접근할 수 있는 특성을 갖는다. 다음의 <그림 1>은 상호 간의 접근 허용 관계를 보인다.



<그림 1> 공유 라이브러리와 일반 사용자 블록들간의 접근 허용 관계

공유 라이브러리로 표현된 s.out 은 일반 사용자 블록과 동일하게 프로그램하면 되나 그 주소 공간이 모든 사용자와 충돌되지 않는 영역이어야 하므로 운영체제에서 공유 공간으로 정의한 영역으로 링크하여야 한다. 이렇게 하여 일반 사용자 블록에서는 자신의 내부 펑션을 호출하는 것과 동일하게 공유 라이브러리 내의 펑션을 호출하여 사용할 수 있다.

이러한 공유 라이브러리를 사용하게 되면, 우선 재사용이 가능한 소프트웨어 요소들을 추출하여 이들을 모듈화 함으로써 부품의 재 사용성을 높이게 되며 공통 자원을 한 곳에서 관리함으로써 일관성 유지가 용이하며 IPC 를 통한 메시지의 전달 부담 대신에 간접적 서브루틴 호출 형태의 빠른 정보 전달이 이루어지는 장점이 있다.

또한 일반 사용자 블록들에 서비스를 제공한다는 관점에서 마치 운영체제나 시스템 소프트웨어와 같은 역할을 수행하나 이들이 커널 모드에서 동작하는 반면 공유 라이브러리는 시스템의 모드로 전이하지 않고 사용자 모드에서 수행함으로써 잘못된 제어로 인한 시스템 손상의 위험 부담을 줄일 수 있어 실시간 시스템의 안정성을 높일 수 있다.

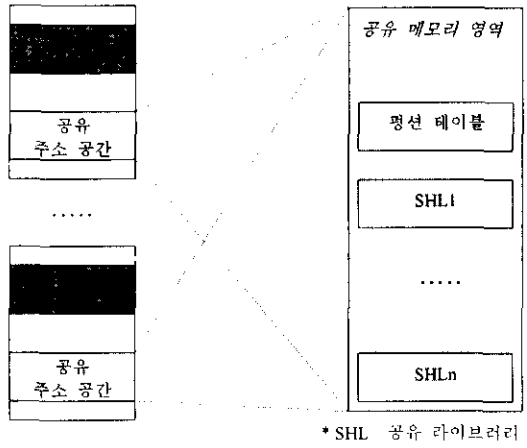
3. 능동적 메모리 개체로서의 동작 구조

UNIX 의 공유 메모리[3]는 그 공간 내에서 프로그램이 동작할 수 있는 능동성이 없으며 데이터가 읽기, 쓰기가 되는 수동적 개체라고 할 수 있다.

본 논문에서 제시하는 공유 라이브러리는 UNIX 의 공유 메모리 성격을 띄지만 메모리의 내용을 읽고 쓰는 것에서 더 나아가 메모리에 쓰여진 내용이 동작의 주체가 되어 실행을 할 수 있는 특성을 가지고 있다. 또한 실행 시 공유 메모리 개체를 생성하고 이후 이를 사용하는 UNIX 의 공유 메모리와는 달리 운영체제 내에서 이러한 기능을 수행하므로 응용 프로그램에서는 간단히 이를 사용하기만 하면 된다. 본 장에서는 이러한 능동적 실행 개체가 되도록 하기 위하여 운영체제와 공유 라이브러리 블록, 이를 사용하는 일반 사용자 블록 각각에서 수행하는 기능과 준수해야 하는 인터페이스 및 제약 사항들을 기술한다.

3.1 운영체제에서의 제공 기능

운영체제에서는 우선 시스템 내 전체 주소 공간 중에서 공유 공간을 설정하고 이 공간이 외부 사용자 블록들에서도 접근이 가능하도록 제어를 하여야 하는데 이를 위한 구조는 다음의 <그림 2>와 같다.



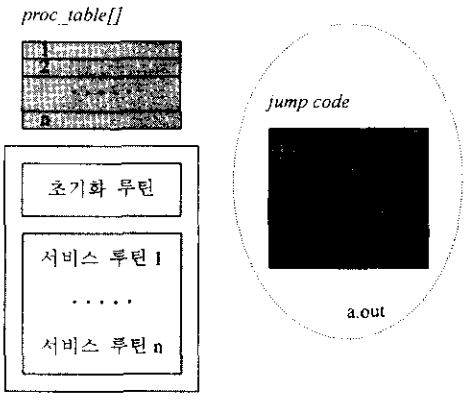
<그림 2> 운영체제에서 구현하는 일반 블록과 공유 라이브러리의 주소 공간 구조

설정된 공간에 대하여는 사용자 모드에서 읽기 쓰기 뿐만 아니라 실행이 가능한 모드로 설정을 해 주어야 하는데 이는 소프트웨어적인 제어만으로는 되지 않으며 하드웨어 장치인 메모리 관리 유니트(MMU: Memory Management Unit)를 구동하여 제어가 가능하다. 이때 공유 공간은 가상 주소 영역으로 그 경계가 광범위하게 설정되어 있지만 실제적으로 공유 블록이 로딩될 때 할당된 영역만을 맵핑한다. 가상 번지와 할당된 물리 번지를 맵핑한 후 이전에 로딩된 블록들에서도 인식이 가능하도록 만들어 주며 이후 로딩되는 블록들도 공유가 가능하도록 관리한다. 또한 이 공유 블록이 메모리에서 삭제될 때는 이러한 공유 정보도 무효화 시킨다. 또한 공유 라이브러리의 펑션들을 분리하여 관리함으로써 잘못된 데이터 접근으로부터 펑션 테이블에 대한 보호 기능 및 펑션 조사를 용이하게 한다.

이 평션 테이블은 시스템 내 사용 가능한 수를 예측하여 충분한 크기로 가상 공간을 설정하여 물리 메모리를 할당하여 맵핑을 하여 둔다. 여기서 사용자 주소 공간과 공유 주소 공간은 실제로 할당되는 영역 만큼만 맵핑이 된다.

3.2 공유 라이브러리의 구성

공유 라이브러리는 타 블록에 서비스를 제공하는 기능을 수행하는 특수 블록으로 그 구성은 초기화 루틴과 일련의 서비스 루틴들의 집합으로 이루어진다. 초기화 루틴에서는 자신의 평션 테이블의 시작 주소를 설정하여야 하며 이를 기점으로 모든 서비스 루틴들을 등록하여 평션 테이블 배열의 인덱스로 접근하여 이동할 수 있도록 한다. 이러한 서비스 루틴들은 여러 사용자 프로세스에서 동시에 접근이 가능하므로 재실행이 가능한 구조이어야 하며 공통 데이터를 관리하는 부분은 임계 영역으로 설정하여야 한다. 또한 이 공유 블록은 사용자 블록과는 같이 링크되지 않으므로 사용자 블록에게 자신과는 별도로 서비스 루틴으로 이동할 수 있는 점프 코드를 제공해야 한다. 이 점프 코드는 사용자 블록과 함께 링크 되어 평션 테이블의 인덱스를 사용하여 간접적으로 서비스 루틴을 호출하게 된다. 이러한 공유 라이브러리의 구성 요소는 다음의 <그림 3>과 같다.

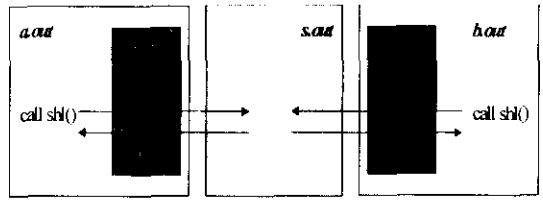


<그림 3> 공유 라이브러리의 구성 요소

*proc_table[]*은 공유 라이브러리 내의 서비스 평션들의 주소를 저장하고 있는 배열이며 각 평션에 해당하는 인덱스는 상수 값으로 정의하여 공유 라이브러리와 점프 코드에서 공통으로 사용한다. 초기화 루틴에서는 자신의 서비스 평션들을 이 proc_table[]에 등록하는 기능을 수행해야 한다. 초기화 루틴과 서비스 루틴들을 컴파일하고 지정된 주소로 링크하여 하나의 실행 모듈로 구성한다. 이때 공유 라이브러리의 프로그램에 대한 제약 사항은 전혀 없으며 일반 사용자 블록과 동일하게 사용자 모드로 동작되며 운영체제의 모든 시스템 콜을 사용할 수 있다. 단, 각 서비스 평션 내에 임계 영역이 있을 수 있는 데 이를 위하여 초기화 루틴에서는 세마포를 생성하여야 하며 이후 서비스 평션들에서는 생성된 세마포로 임계영역 처리를 하여야 한다. 그림의 점프 코드는 공유 라이브러리와는 별개로 공유 라이브러리를 사용하는 일반 응용 블록에 링크 되는데 실제적으로 사용자가 호출하는 인터페이스 평션이 이 점프 코드 내에 존재하여 해당 평션을 호출할 수 있게 된다.

3.3 공유 라이브러리의 호출

운영체제나 공유 라이브러리와는 달리 일반 사용자 블록은 간단히 마치 자신의 내부 프로시저를 호출하는 것처럼 서비스 루틴들을 호출하면 된다. 이때 일반 프로시저 호출과 동일하게 파라메타 전달과 리턴 값 전달이 이루어진다. 단 사용자 프로그램 컴파일 후 공유 라이브러리용 점프 코드를 같이 링크하여 실행 모듈을 만들어야 한다. 사용자 블록이 실행될 때 공유 라이브러리가 호출되는 구조는 다음의 <그림 4>와 같다. 즉, 사용자 블록은 공유 라이브러리를 직접 호출하는 것이 아니라 자신에게 링크되어 있는 작은 라이브러리인 점프 코드를 호출하고 이 점프 코드 내에서 해당 공유 라이브러리로 이동하게 되는 것이다. 리턴 값도 역시 동일한 경로로 전달된다.



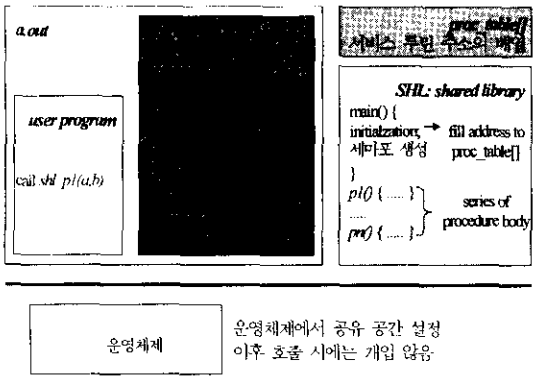
<그림 4> 공유 라이브러리의 호출 경로

3.4 공유 라이브러리와 운영체제 및 사용자 블록과의 인터페이스

위의 3 절에서 운영체제, 공유 라이브러리, 사용자 블록 각각에서 행하는 기능들에 대하여 기술하였으며 본 장에서는 이들이 결합되어 동작되는 전체 구조에 대하여 기술한다. 다음의 <그림 5>는 각 기능 모듈의 위치와 역할을 보이며 공유 라이브러리를 사용하기 위한 프로그래밍 지침을 보인다.

우선 운영체제는 공유 라이브러리의 실행환경을 설정하고 공유 블록이 메모리에 로딩되거나 삭제될 경우에 그 기능이 수행되며 실제 동작 중에는 개입하지 않는다. 즉, 일반 사용자에서 공유 라이브러리를 호출하여 기능 수행 후 리턴 값을 받기까지의 전 과정이 운영체제의 도움 없이 사용자 모드에서 수행이 가능하다. 그 과정을 보면 우선, 사용자 프로그램에서 shl_p1()이라는 공유 라이브러리를 호출하게 되면 자신에 링크되어 있는 점프 라이브러리로 가게 되고, 이 점프 라이브러리 내에서는 평션 테이블의 인덱스로 참조된 평션 엔트리로 이동하게 된다. 이때 공유 데이터의 일관성

유지를 위하여 임계영역을 설정하여야 하는데 이를 위하여 초기화 기능에서 *sema create()*와 같은 운영체제 시스템 콜로 세마포를 생성하여야 하고 임계영역 전후로 세마포 획득(예, *sema_take()*), 양도(예, *sema_give()*)의 시스템 콜을 사용하여야 한다. 수행 후 리턴 값도 역시 점프 코드를 경유하여 전달된다.



<그림 5> 공유 라이브러리를 위한 프로그래밍 지침

4. 적용 실험

본 논문에서 제안한 공유 라이브러리 기법은 TDX-10 전전자 교환기의 운영체제인 CROS(Concurrent Real-Time Operating System)[4]에 적용되어 사용되었으며, ATM 교환기인 HANBit ACE 시스템의 운영체제인 SROS(Scalable Real-Time Operating System)[5]와 소프트웨어 패키지에 적용되어 성공적으로 사용 중에 있다. 64MB 의 메인 메모리 상에서 수행되는 HANbit ACE 시스템에서 실행하는 경우 30MB 정도의 일반 사용자 메모리 영역에서 3MB 정도의 메모리를 절약하게 되어 약 10%의 효율이 높아짐을 보았다. 공유 서비스 종선과 데이터가 많아지면 이 효율도 비례해서 높아질 것이다. 또한 superSPARC CPU 를 장착한 상기 시스템에서 일반적인 공유 메모리의 처리 성능을 100%으로 볼 때 공유 라이브러리의 처리 성능은 97%로서 능동 개체의 호출로 인한 성능 저하가 거의 없었다. 또한 IPC 를 통한 메시지 전달 성능[6]과 비교할 때 3 배 이상의 높은 효율을 보였다. 이는 다음의 <표 1>과 <표 2>에 나타나 있다.

<표 1> 공유 메모리 처리 대비 공유 라이브러리 처리 성능 백분율 표

공유 메모리 처리	공유 라이브러리 처리
100%	97%

<표 2> 메시지 전달 대비 공유 라이브러리 처리 성능 백분율 표

IPC 메시지 전달	공유 라이브러리 처리
100%	31%

현재는 powerPC 750 CPU 를 위한 SROS 인 powerSROS[7]에 적용하여 구현 중에 있다.

5. 결론

본 논문에서는 대형 실시간 시스템의 응용 소프트웨어 개발을 위하여 메모리 자원을 효율적으로 사용하면서 정보 전달 속도를 빠르게 할 수 있는 방법으로 능동적 메모리 개체로서의 공유 라이브러리 기법을 제안하였다. 공유 라이브러리는 수동적인 개체인 일반 공유 메모리 개념에 능동적으로 동작할 수 있는 구조를 추가하였으며 연결되는 프로그램에 실행 이미지가 밀 결합되는 일반적인 라이브러리와 달리 사용하는 프로그램과는 무 결합 구조로서도 실행 가능한 구조적 장점을 갖는다. 이는 운영체제에서 공유 메모리 공간을 능동적 개체로 설정하는 기능을 제공하고, 공유 라이브러리 블록에서는 간접적인 호출이 가능한 구조를 구현함으로써 가능하였다. 이로써 메인 메모리의 효율을 10% 이상 높일 수 있었으며, 일반적인 공유 메모리 처리의 97%의 성능을 보여 능동 개체 호출로 인한 부담은 거의 없음을 보았으며 메시지 전달보다는 3 배 이상의 높은 처리 성능을 보임으로써 실시간 시스템의 자원 사용의 효율성 및 성능 향상에 기여하였다. 또한 제안된 기법은 하부 하드웨어 플랫폼의 변경 시에도 운영체제에서 제공하는 기능의 일부를 변경하여 적용할 수 있도록 구성되어 우수한 이식성을 갖는다. 향후 본 연구를 발전시켜 범용적으로 사용 가능하도록 하기 위하여 일반적인 상용 운영체제 상에서 본 기법을 적용할 경우의 구현 방법에 대한 연구가 필요하다.

참 고 문 헌

[1] H.H.Kim, S.I.Jun, Y.J.Cha, J.H.Cho, "HLRP: A High-performance Light-weight Real-time Protocol", IEICE'95 Proceeding, pp.467-470, Aug. 7-12, 1995.

[2] 정부금, 차영준, 전성익, 조주현, "분산 실시간 시스템을 위한 멀티 유저 프로그래밍의 설계 및 구현", 전자공학회 충청지부 하계학술대회 논문집, pp. 40-43, May 30, 1997.

[3] Prabhat K. Andleigh, "UNIX system architecture", Prentice Hall, pp.96-98, 1990.

[4] 강석열, 윤용익, 조주현, 정부금, 정영조, "대용량 교환기를 위한 CONCURRENT REALTIME OPERATING SYSTEMS (CROS) 의 개발", JCTC'88, pp. 20-24, Oct. 20-22, 1988.

[5] 정부금, 차영준, 장형규, 이은향, 김형환, 전성익, 조주현, 최완, "SROS: ATM 교환기를 위한 스케일러블 구조의 실시간 운영체제", COMSW'97 논문집, pp.425-429, Jul. 24-26, 1997.

[6] 박영호, 김형환, 정부금, 전성익, 임동선, "ATM 교환기를 위한 실시간 운영체제에서 프로세서간 통신기능 성능 분석", SWCC'98 하계컴퓨터통신워크샵 논문집, pp.231-234, Aug. 20-21, 1998.

[7] http://circle.etri.re.kr/