

KDSM (KAIST Distributed Shared Memory) 시스템의 설계 및 구현

Design and Implementation of KDSM (KAIST Distributed Shared Memory) System

이상권(Sang-Kwon Lee) 윤희철(Hee-Chul Yoon)

한국과학기술원 전산학과 컴퓨터구조 연구실

2000년 7월 15일

요약

본 보고서에서는 프로토타입 분산 공유메모리 시스템인 KDSM (KAIST Distributed Shared Memory) 시스템에 관해서 설명한다. KDSM 시스템의 특징은 페이지 기반 무효화 프로토콜(page-based invalidation protocol)과 홈 기반 프로토콜(home-based protocol)을 기반으로, HLRC(Home-based Lazy Release Consistency) 및 ScC(Scope Consistency) 두 가지 memory consistency model을 제공한다는 것이다. KDSM 시스템은 100 Mbps Fast Ethernet으로 연결된 여러 대의 PC로 구성된 클러스터 시스템에서 구현되었다. KDSM 시스템은 Linux 커널을 전혀 수정하지 않은 사용자 수준 프로세스로 구현되었고, 기반 통신 구조로 TCP/IP를 사용한다.

⁰본 연구는 국가지정연구실사업의 지원을 받았다.

목 록

제 1 장 서론	4
제 2 장 실행 및 프로그래밍 환경	5
제 1 절 설치	5
제 2 절 실행환경	7
제 3 절 프로그래밍 환경	8
제 3 장 메모리 관리	11
제 1 절 메모리 구조	11
제 2 절 메모리 맵핑	12
2.1 mmap() 함수	12
2.2 공유메모리의 맵핑	13
2.3 공유메모리 맵핑을 위한 자료 구조	14
2.4 SIGSEGV 시그널 핸들러	15
2.5 캐쉬 페이지 상태 전이도	17
제 3 절 메모리 할당	18
제 4 장 통신 구조	20
제 1 절 메시지 구조	20
제 2 절 프로세스간 통신	21

2.1	커백션 생성	21
2.2	메시지의 전송	23
2.3	SIGIO 시그널 핸들러	24
제 5 장	Home-based Lazy Release Consistency의 구현	26
제 1 절	Home-based Lazy Release Consistency	26
제 2 절	Lock의 구현	28
2.1	상호배제(Mutual Exclusion)	28
2.2	메모리 일관성(Consistency)	31
2.3	알고리즘	35
제 3 절	Barrier의 구현	38
제 6 장	Scope Consistency 프로토콜의 구현	41
제 1 절	Scope Consistency [1]	41
제 2 절	Lock의 구현	42
2.1	Lock 프로토콜의 실행 예	42
2.2	중첩된 lock의 구현	45
제 3 절	Barrier의 구현	49
제 7 장	성능 측정	51
제 8 장	결론	54

제 1 장

서론

본 보고서에서는 구현된 프로토타입 DSM 시스템인 KDSM(KAIST Distributed Shared Memory) 시스템에 관해서 설명한다.

KDSM 시스템은 여러 대의 PC를 100Mbps Fast Ethernet으로 엮은 클러스터 시스템 상에서 실행된다. 각 PC들은 두 개의 500 Mhz Pentium III 프로세서를 탑재하고 있으며, 운영체제로 Linux 2.2.3SMP를 사용한다. KDSM 시스템은 사용자 수준 프로세스(user-level process)로 동작한다. 기반 통신 구조로는 TCP/IP를 사용해서 프로세스 간 통신을 수행한다. 캐쉬 일관성 프로토콜은 페이지 기반 무효화 프로토콜(page-based invalidation protocol)과 홈 기반 프로토콜(home-base protocol)을 기반으로 해서 다중 읽기 다중 쓰기(multiple reader multiple writer) 프로토콜을 사용한다. 또한 KDSM 시스템은 HLRC(Home-base Lazy Release Consistency) [3]와 ScC(Scope Consistency) [1] 두 개의 memory consistency model을 제공한다.

본 보고서의 구성은 다음과 같다. 2장에서는 공유메모리 구조, 메모리 맵핑, 메모리 할당 방법에 대해서 설명한다. 3장에서는 분산 공유메모리의 기반 통신 환경에 대해서 설명한다. 4장에서는 구현된 시스템의 실행 환경 및 프로그래밍 환경에 대해서 설명한다. 5장과 6장에서는 각각 HLRC, ScC 프로토콜과 그 구현 사항을 자세히 살펴본다. 7장에서는 구현된 시스템의 성능 측정 결과를 설명한다. 마지막으로 8장에서 결론을 맺는다.

제 2 장

실행 및 프로그래밍 환경

이 장은 KDSM 시스템의 사용자의 입장에서 설치 및 실행 방법 그리고 프로그래밍 인터페이스에 대해 설명한다.

1 설치

현재 개발된 시스템은 linux kernel 2.2.13-smp 버전에서 개발되고 테스트되었으나 다른 버전의 리눅스 환경에서도 잘 동작할 것이다. tar로 묶여있는 파일을 풀게 되면 kdsmdir 디렉토리가 생성된다. 이 디렉토리는 소스파일이 들어있는 src/ 디렉토리와 library파일이 생성되는 lib/ 디렉토리, 응용 프로그램이 포함(include)시켜야 할 헤더 파일이 들어있는 include/ 디렉토리, 문서 및 논문들이 들어있는 docs/ 디렉토리, KDSM 시스템을 이용한 응용 프로그램들이 들어있는 apps/ 디렉토리와 README 파일로 구성되어 있다. 그림 2.1 는 KDSM 시스템의 전체 디렉토리 구조를 나타낸다.

lib/ 디렉토리에서 make 를 하면 libHLRC.a 와 libSCC.a 의 두개의 라이브러리가 생성된다. 이는 각각 HLRC 프로토콜과 SCC 프로토콜로 동작하는 DSM 시스템 라이브러이인데 사용자는 원하는 라이브러리를 링크하여 사용하면 된다. 두 라이브러리 모두 동일한 프로그래밍 인터페이스(API)를 제공하므로 어떤 라이브러리를 쓰더라도 프로그램이 바뀔 필요가 없다. KDSM 시스템은 NFS 환경과 그렇지 못한 환경을 모두 지원한다. 기본 설정은 NFS 환경에서 동작하도록 되어

```

kdsm/
|-- apps
|   |-- WATER
|   |   |-- Makefile
|   |   .
|   |-- TSP
|   .
|   .
|
|-- doc
|-- include
|   '-- dsm.h
|-- lib
|   |-- Make.common
|   |-- Makefile
|   |-- Makefile.HLRC
|   |-- Makefile.SCC
|   |-- libHLRC.a
|   '-- libSCC.a
'-- src
    |-- comm.c
    |-- comm.h
    |-- diff.c
    |-- diff.h
    |-- exit.c
    |-- global.h
    |-- hlrc.c
    |-- hlrc.h
    |-- hlrc_wtnt.c
    |-- hlrc_wtnt.h
    |-- init.c
    |-- memory.c
    |-- memory.h
    |-- msg.c
    |-- msg.h
    |-- scc.h
    |-- scc_lock.c
    |-- scc_lockmgr.c
    |-- scc_wtnt.c
    |-- stat.h
    |-- utility.c
    '-- utility.h

```

그림 2.1: KDSM 시스템의 디렉토리 구조

있으나 NFS가 설치되지 않은 환경에서는 lib/ 디렉토리의 Make.common 파일에서 NFS 지원 플래그를 빼면된다.

```
CFLAGS = -c -DNFS -D$(PROT)
```

NFS 환경이 아닌 경우 KDSM은 원격 호스트의 홈디렉토리에 프로그램의 실행파일을 자동으로 복사한후 실행한다. 만일 응용 프로그램이 별도의 데이터파일이 필요하다면 각 호스트의 홈 디렉토리에 사용자가 직접 복사해 주어야 한다.

KDSM 시스템은 원격 프로세스를 수행하기 위해 rsh(1) 프로그램을 이용한다. rsh을 사용하기 위해서 사용자는 프로세스를 수행할 각호스트의 홈디렉토리의 .rhosts 파일을 수정하여 서로간에 rsh을 수행할 수 있도록 하여야 한다. .rhosts 파일의 설정에 대하여는 manual 페이지를 참조하길 바란다.

2 실행환경

KDSM 시스템을 이용하여 프로그램을 수행하기 위해서는 수행하는 프로그램의 실행파일이 있는 디렉토리에 .config 라는 설정파일을 만들어주어야 한다. 이 설정파일은 프로세스를 실행시킬 노드에 대한 정보를 기록하는 파일로서 다음과 같이 매 라인마다 호스트의 이름을 기록한다. 주석으로 처리하려면 맨 앞에 '#' 문자를 쓰면 해당 라인을 무시한다.

```
can1
can2
can3
```

그림 2.2: .config 파일의 예

위의 예는 can1, can2, can3 라는 세 호스트에 각각 하나씩의 프로세스를 수행시키겠다는 뜻이다. 하나의 호스트에서 여러 프로세스를 수행시키려면 단순히 이 설정파일에서 같은 호스트 이름을 여러번 쓰면 된다. 설정 파일을 만들때 주의할 것은 맨 첫줄의 호스트는 반드시 사용자가 프로

그램을 수행시키는 호스트여야 한다는 점이다. 2.2 의 예에서 만일 can2 라는 호스트에서 작업을 한다면 can2가 맨 첫줄에 있어야 한다.

3 프로그래밍 환경

프로토타입 DSM 시스템은 응용 프로그램 개발을 위해 다양한 API들을 제공한다. API 인터페이스는 dsm.h 파일에 정의되어 있으며, 각 응용 프로그램은 이 파일을 포함해야 한다. 프로토타입 DSM 시스템이 응용 프로그램에 제공하는 API는 다음과 같다.

- `DsmInit(argc, argv)`

DSM 시스템을 초기화시킨다. 이 함수는 모든 응용 프로그램의 첫부분에서 호출되어야 한다. `DsmInit()`은 환경 설정 파일을 읽어서 각 호스트에 응용 프로그램을 실행시킨다.

- `DsmAlloc(size)`

`size` 크기 만큼의 공유메모리를 할당한다.

- `DsmAllocAt(size, pid)`

`size` 크기 만큼의 공유메모리를 주어진 프로세스(`pid`)에 할당한다.

- `DsmAllocBlock(size, blocksize)`

`size` 크기 만큼의 공유메모리를 할당하는데, `blocksize` 만큼 프로세스들을 돌아가면서 할당한다.

- `DsmAllocBlockAt(size, blocksize, pid)`

`size` 크기 만큼의 공유메모리를 할당하는데, 주어진 프로세스(`pid`)에서 부터 시작해서 `blocksize` 만큼 프로세스들을 돌아가면서 할당한다.

- `DsmLock(lockid)`

주어진 `lock`에 대해서 `lock`을 얻는다(`acquire`).

- `DsmUnlock(lockid)`

주어진 lock에 대해서 lock을 해제한다(release).

- `DsmBarrier()`

전역적인 동기화를 위해서 barrier 연산을 수행한다.

- `DsmExit()`

모든 프로세스를 동기화 한 후 DSM 시스템을 종료한다.

- `DsmGetPid()`

현재 응용 프로그램이 실행되고 있는 DSM 시스템의 프로세스 번호를 알아낸다. 프로세스의 번호는 .config 파일에 기록되어 있는 순서로 0부터 n-1 까지 할당된다.

- `DsmGetProcNum()`

DSM 시스템에서 실행 중인 프로세스 수를 알아낸다. 이것은 .config 파일에 기록되어 있는 호스트의 갯수와 일치한다.

- `DsmGetNodeNum()`

DSM 시스템이 운용되는 전체 노드의 갯수를 반환한다. 노드의 갯수는 프로세스의 갯수와 다를 수 있는데 이것은 하나의 노드에서 여러개의 프로세스가 동작할 수 있기 때문이다. 만일 하나의 호스트에서 반드시 하나의 프로세스만이 동작한다고 하면 `DsmGetProcNum()` 과 동일한 값을 돌려주게 된다.

KDSM 라이브러리를 사용하기 위해서는 `dsm.h` 파일을 프로그램내에 포함해야 한다. KDSM 라이브러리는 SPMD 프로그래밍 모델 (동일한 프로세스가 각각 다른 부분의 공유 메모리를 이용하는 프로그래밍 모델) 을 제공한다. 그림 2.3 은 KDSM 라이브러리를 이용하는 일반적인 프로그램의 구조를 나타낸다. 각 프로세스는 제일먼저 `DsmInit(argc, argv)`를 호출하여 KDSM 라이브러리를 사용하기 위한 내부 초기화 과정을 거친다. 그 다음으로는 `DsmAlloc()`, `DsmAllocAt()`, `DsmAllocBlock()`, `DsmAllocBlockAt()` 등을 이용하여 공유메모리를 할당한다. 공유메모리를

할당 후에는 반드시 `DsmBarrier()` 를 호출하여 모든 프로세스가 공유메모리를 할당했음이 보장되어야 한다. `Worker()` 루틴은 실제로 공유메모리를 이용하여 동작하는 루틴이다. 각 프로세스의 `Worker()` 루틴은 `DsmLock()`, `DsmBarrier()` 등을 이용하여 공유메모리 상에서 상호 협력하에 동작한다. `Worker()` 루틴이 끝난 후에는 `DsmExit()` 를 호출하여 모든 `Worker()` 프로세스가 끝나는 것을 기다린 후 종료한다.

```
#include <dsm.h>

char *ptr1, *ptr2, ... ;

void Worker()
{
    /* 공유 메모리 상에서 동작하는 프로그램. DsmLock(), DsmUnlock(),
       DsmBarrier() 등에 의해 동기화를 한다. */
    .
    .
}

int main(int argc, char* argv[])
{
    /* KDSM의 초기화. 원격 프로세스 생성 */
    DsmInit(argc, arg);

    /* 공유 메모리 할당 */
    ptr1 = DsmAlloc(size1);
    ptr2 = DsmAllocAt(size2, pid);
    .
    .
    DsmBarrier();

    /* worker 프로세스 시작. 여러 개의 프로세스가 같은 공유 데이터
       영역에서 동작함 */
    Worker();

    /* KDSM 종료 */
    DsmExit();

    return 0;
}
```

그림 2.3: KDSM 응용프로그램의 일반적인 구조

제 3 장

메모리 관리

1 메모리 구조

KDSM 시스템은 NUMA(Non-Uniform Memory Access) 형태의 메모리 구조를 가진다 (그림 3.1). 공유 페이지들은 미리 지정된 홈 노드들을 가지고, 공유 페이지들의 홈 노드는 모든 노드들에 골고루 분산된다. 각 노드들은 원거리 페이지를 저장하기 위해 소프트웨어 캐쉬를 사용한다. 홈 노드에 있는 페이지에 대한 참조는 지역적으로 해결되나, 홈이 아닌 페이지에 대한 참조는 폴트가 발생하게 되어 홈 노드로부터 페이지를 읽어 와서 캐쉬에 저장한다. 캐쉬된 페이지들은 읽기전용 상태(RO), 읽고쓰기 상태(RW), 무효화 상태(INV) 상태 중 하나가 된다.

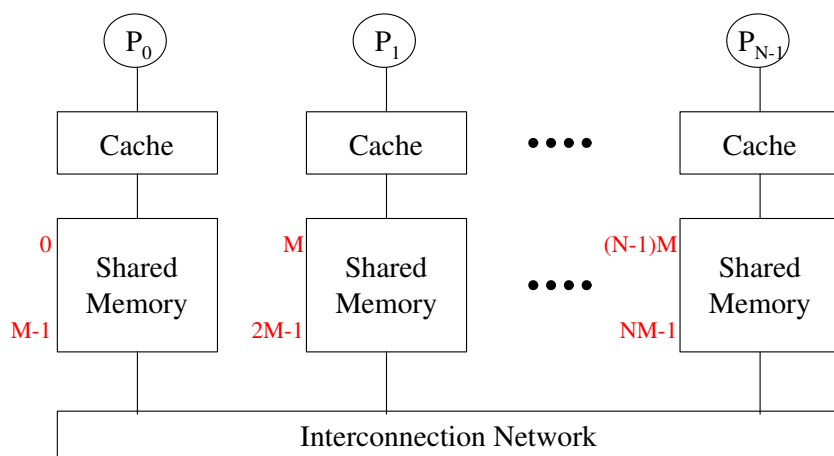


그림 3.1: 메모리 구조

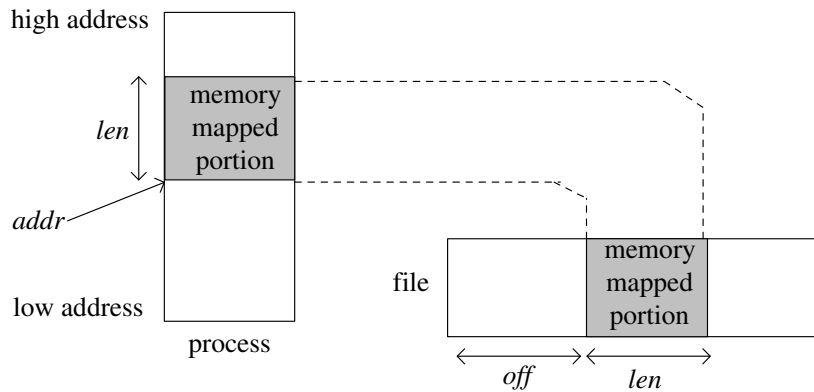


그림 3.2: memory mapped file의 예

2 메모리 맵핑

KDSM 시스템은 UNIX의 `mmap()` 시스템 호출을 사용해서 공유메모리를 맵핑한다.

2.1 `mmap()` 함수

`mmap()`은 memory mapped IO를 사용해서 디스크 상의 파일을 메모리 상의 버퍼로 맵핑한다. 버퍼에서 어떤 바이트를 읽으면 파일의 해당 바이트가 읽히고, 비슷하게 버퍼에 어떤 값을 저장하면 해당 바이트가 파일에 자동으로 저장된다. 그림 3.2는 memory mapped file의 예를 보여준다.

`mmap()` 함수의 프로토타입은 다음과 같다:

```
caddr_t mmap(caddr_t addr, size_t len, int prot, int flag, int filedes, off_t off);
```

`addr`은 맵핑될 메모리 영역의 시작 주소를 지정하고, `len`은 맵핑할 바이트 수를 지정한다. `filedes`는 맵핑할 파일을 나타내는 파일 디스크립터이고, `off`는 맵핑할 바이트들의 파일 내 상대 주소이다. `prot`는 맵핑될 영역에 대한 보호 모드를 지정하는데, 다음과 같은 값을 가질 수 있다.

<i>prot</i>	설명
PROT_READ	읽기 가능
PROT_WRITE	쓰기 가능
PROT_EXEC	실행 가능
PROT_NONE	접근 불가능

*flag*는 맵핑된 영역에 대한 다양한 속성을 지정한다.

- MAP_FIXED

`mmap()`의 리턴 값이 *addr*과 같아야 함을 의미한다.

- MAP_SHARED

store 연산이 맵핑된 파일을 수정하도록 한다. 즉, store 연산이 파일에 대한 write와 동등하다. MAP_SHARED 또는 MAP_PRIVATE 둘 중에 하나가 지정되어야 한다.

- MAP_PRIVATE

맵핑된 메모리 영역에 대한 store 연산이 맵핑된 파일에 대한 복사본을 생성하게 한다. 맵핑된 영역에 대한 이후의 모든 참조는 복사본을 참조한다.

맵핑된 메모리 영역과 관련해서 다음 두 가지 경우에 SIGSEGV 시그널이 발생한다.

- 현재 사용할 수 없는 메모리를 접근했을 경우
- 읽기 전용으로 지정한 메모리 영역에 대해서 쓰기를 시도했을 경우

2.2 공유메모리의 맵핑

공유메모리의 맵핑은 그림 3.3과 같이 공유메모리의 주소 공간을 UNIX의 특수 파일인 `/dev/zero`로 맵핑함으로써 이루어진다. `/dev/zero` 파일에 맵핑하는 것은 맵핑된 길이만큼의 길이를 가지면서 0으로 초기화된 메모리 영역을 생성한다. `STARTADDR`는 전체 주소 공간에서 공유메

```

fd = open("/dev/zero", O_RDWR, 0);
...
mmap(STARTADDR + GlobalOffset, len, PROT_READ, MAP_PRIVATE|MAP_FIXED, fd, 0);

```

그림 3.3: 공유메모리 맵핑 방법

모리로 사용할 부분의 시작 주소이다. GlobalOffset은 현재까지 할당된 공유메모리 양을 표시한다. 공유 페이지는 초기에 홈 노드에 읽기 전용 상태로 맵핑되는데 이것은 나중에 write가 발생했을 때 그것을 검출하기 위해서 이다.

그림 3.4는 각 노드마다 M 바이트를 공유메모리로 지정하고, 프로세서 0번부터 N-1번까지 순차적으로 공유메모리를 맵핑되었을 때의 예를 보여준다. 실제 공유메모리를 4 KB 크기의 페이지 단위로 할당이 된다.

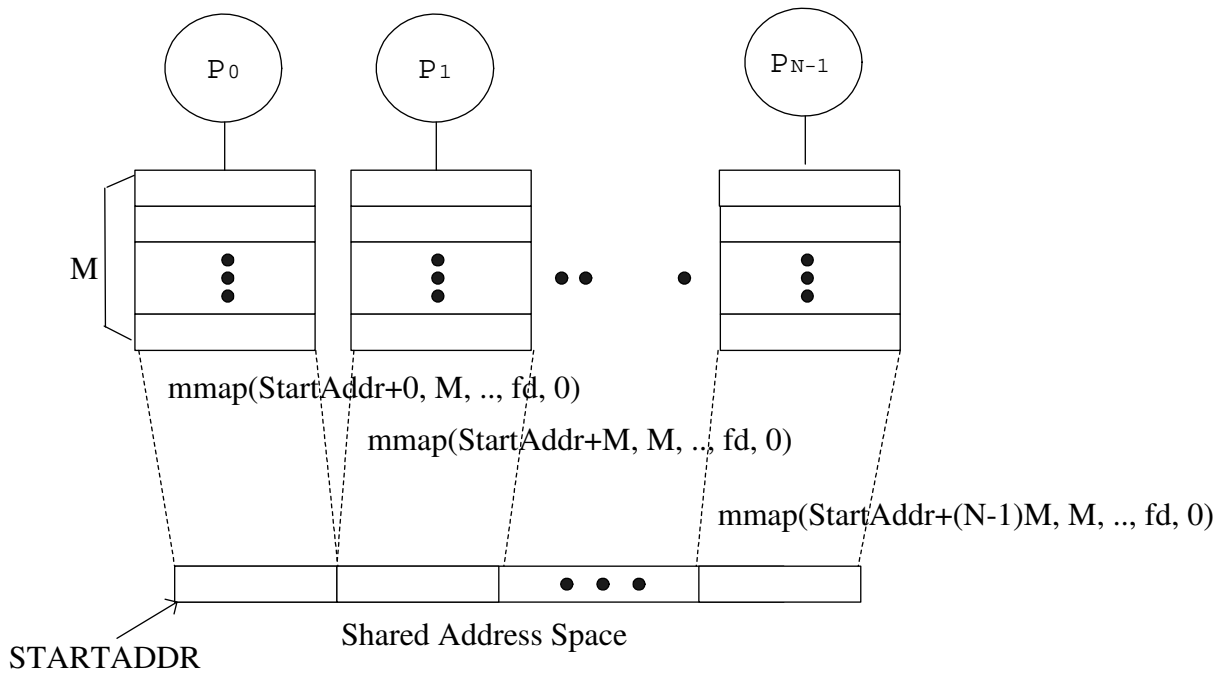


그림 3.4: 공유메모리 맵핑의 예

2.3 공유메모리 맵핑을 위한 자료 구조

공유메모리를 유지하기 위해서 다음과 같은 자료 구조가 사용된다:

- gProc [MAXPROCS]

각 노드에 실행되는 DSM 프로세스에 관한 정보를 유지한다. 각 항목은 프로세스 번호에 의해서 선택되고, 해당 프로세스에 할당된 공유메모리의 크기(size) 필드를 가진다.

- gPage [MAXLOCALPAGE]

각 홈 노드에 할당된 공유메모리에 해당하는 페이지 정보를 유지한다. 각 항목은 공유메모리 페이지의 주소(addr) 필드와 해당 페이지가 수정되었는지를 표시하는 플래그(written) 필드로 구성된다.

- gCache [MAXCACHEPAGE]

홈이 아닌 원거리 페이지들을 저장하는 캐쉬 정보를 유지한다. 각 항목은 캐쉬에 저장된 페이지의 공유메모리 주소(addr) 필드, 페이지의 현재 상태(state), 페이지가 수정되었을 때 생성하는 twin을 가르키는 포인터(twin)로 구성된다. 페이지의 상태 필드는 초기에 맵핑이 설정되지 않은 상태(UNMAP)에서 읽기전용 상태(RO), 읽고쓰기 상태(RW), 무효화 상태(INV) 상태 중 하나가 된다.

- gGPage [MAXGLOBALPAGE]

공유메모리 전체 페이지에 대한 정보를 유지하고, 각 항목은 공유메모리의 페이지 번호에 의해 선택된다. homepid 필드는 해당 페이지의 홈 프로세스를 저장한다 (KDSM 시스템은 공유메모리를 특정 프로세스에서 할당할 수 있기 때문에 이 필드가 반드시 필요하다). pagei 필드는 홈 프로세스에만 의미가 있는 값으로 해당 페이지의 지역 정보를 참조할 때, 즉, gPage[]을 접근할 때 그 색인으로 사용한다. cachei 필드는 원거리 페이지에 대한 캐쉬 정보를 참조할 때, 즉 gCache[]를 접근할 때 그 색인으로 사용한다.

2.4 SIGSEGV 시그널 핸들러

그림 3.5는 SIGSEGV 시그널 핸들러를 위한 알고리즘을 의사 코드 형식으로 보여준다. 2.1 절에서 언급한 바와 같이 SIGSEGV 시그널은 두 가지 경우에 발생한다. SIGSEGV 시그널이 발생하면 해당

Input: fault address

Output: none

Algorithm:

```
if (the fault page is local page) // write fault on local RO page
    set protection mode of the page to PROT_READ|PROT_WRITE;
else // remote page fault
{
    if (write fault on RO page)
    {
        set protection mode of the page to PROT_READ|PROT_WRITE;
        create twin and set cache state to RW;
    }
    else
    {
        if(read or write fault on INV page)
            set protection mode of the page to PROT_READ|PROT_WRITE;
        else // read or write fault on UNMAP page
            map the fault page using mmap() with PROT_READ|PROT_WRITE mode;

        send a GETP to home of the page;

        if (write fault)
        {
            create twin and set cache state to RW;
            wait until GETPGRANT is received;
            copy the fault page into cache;
        }
        else // read fault
        {
            set cache state to RO;
            wait until GETPGRANT is received;
            copy the fault page into cache;
            set protection mode of the page to PROT_READ;
        }
    }
}
```

그림 3.5: SIGSEGV 시그널 핸들러를 위한 알고리즘

시그널 핸들러가 불려지고, 시그널 핸들러는 파라미터로 넘어 온 값을 분석해서 폴트가 난 주소를 알아낸다. 이 주소를 사용해서 각 경우에 다음과 같이 처리함으로써 분산 공유메모리를 유지한다.

- 현재 사용할 수 없는 메모리를 접근했을 때

이 경우는 원거리 페이지를 참조했을 때이다. 시그널 핸들러는 폴트난 페이지를 저장한 공간을 캐쉬에 할당하고 이 페이지로 메모리를 맵핑한 후, 폴트난 페이지의 홈 프로세스에게 GETP 메시지를 전송해서 페이지를 요청한다. GETPGRANT 메시지가 도착할 때까지 대기하다가, 메시지가 오면 폴트 페이지를 캐쉬에 할당된 페이지로 읽어 들인다. 캐쉬에는 접근 형태에 따라서 RO 또는 RW 상태를 기록한다.

- 읽기 전용으로 지정된 메모리 영역에 대해서 쓰기를 시도했을 때

만약 지역 페이지에 대한 쓰기라면 페이지가 수정되었을 표시한 후, 페이지의 보호 모드를 읽고쓰기 모드로 바꾸면 된다. 캐쉬에 저장된 원거리 페이지에 대한 쓰기라면 페이지가 수정되었음을 표시한 후, 페이지에 대한 보호 모드를 읽고쓰기 모드로 바꾼다. 또한 나중에 diff를 생성하기 위해서 twin 생성하고, 캐쉬 상태를 RW로 바꾼다.

2.5 캐쉬 페이지 상태 전이도

그림 3.6는 ScC 프로토콜에서 캐쉬 상태 전이 과정 요약한 것이다 (편의상 UNMAP 상태는 제외하였다.)

INV 상태인 페이지에 대해서 read 연산을 수행하면 SIGSEGV 핸들러에 의해서 홈 프로세스로 GETP 메시지가 전송되어 페이지를 읽고 상태를 RO로 바꾼다. INV 상태인 페이지에 대해서 write 연산을 수행하면 SIGSEGV 핸들러에 의해서 홈 프로세스로 GETP 메시지가 전송되어 페이지를 읽는다. 그 후 twin을 생성해서 원본을 저장하고 캐쉬 상태를 RW 상태로 바꾼다.

RO 상태인 페이지에 대해서 read 연산을 수행하는 것은 폴트가 일어나지 않고, 캐쉬 상태에도 아무런 영향을 미치지 않는다. acq, rel barr 연산이 수행될 때에도 RO 상태인 페이지는 아무런 영향을 받지 않는다. RO 상태인 페이지에 대해서 write 연산을 수행하면 SIGSEGV 시그널이 생성된다. SIGSEGV 핸들러는 twin을 생성해서 원본을 저장한 후 캐쉬 상태를 RW 상태로 바꾼다.

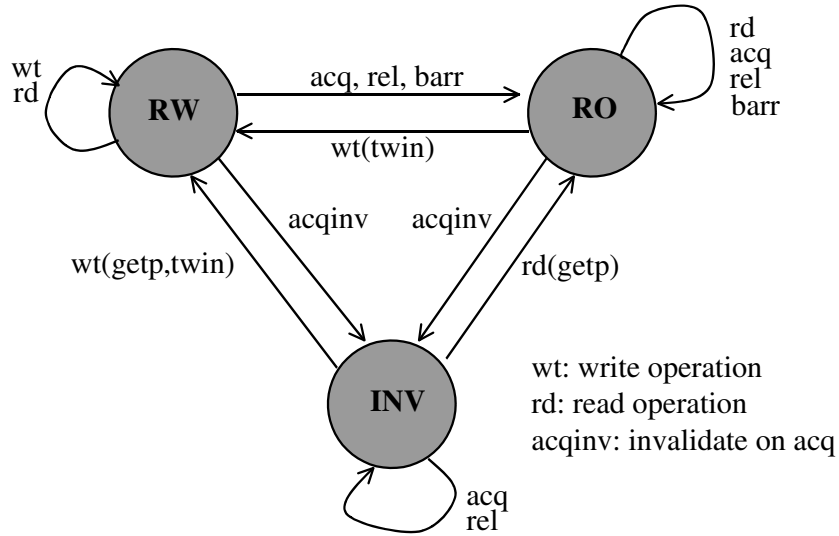


그림 3.6: 캐쉬 상태 전이도

RW 상태인 페이지에 대해서 read나 write 연산을 수행하는 것은 폴트가 일어나지 않고 캐쉬 상태에도 아무런 영향을 미치지 않는다. acq 연산을 수행할 때는 앞에서 언급했듯이 RW 상태를 RO 상태로 바꾸어야 한다. 비슷하게 rel, barr 연산을 수행할 때에도 RW 상태를 RO 상태로 바꾼다.

RO, RW 두 상태 모두 ACQGRANT 메시지와 함께 오는 write notice에 의해서 무효화되면 INV 상태로 바뀐다.

3 메모리 할당

KDSM 시스템을 다음과 같이 네 가지 메모리 할당 함수를 제공한다. 그림 3.7은 각 함수들의 예를 보여준다.

- `DsmAllocAt(size, pid);`

프로세스 pid에서 부터 size 바이트 만큼의 공유메모리를 할당한다. 만약 pid 프로세스에서 필요한 만큼의 공유메모리를 할당 받을 수 없으면 pid + 1, pid + 2, ... 이런 식으로 다음 프로세스에서 할당받을려고 시도한다.

- `DsmAlloc(size);`

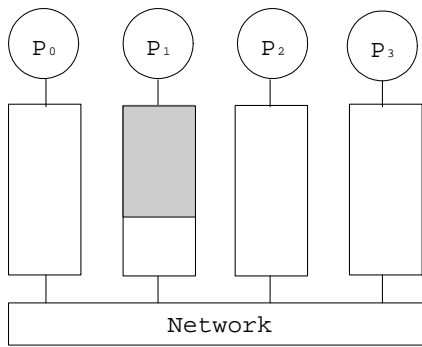
`DsmAllocAt(size, 0)`에 해당한다. 즉, 프로세스 0번부터 공유메모리를 할당 받을려고 시도한다.

- `DsmAllocBlockAt(size, blocksize, pid);`

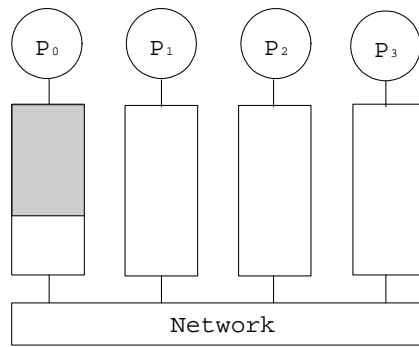
프로세스 `pid`에서 부터 시작해서 round-robin 방식으로 각 프로세스에 `blocksize` 바이트만 큼씩 공유메모리를 할당해서, 전체 `size` 바이트를 할당한다.

- `DsmAllocBlock(size, blocksize);`

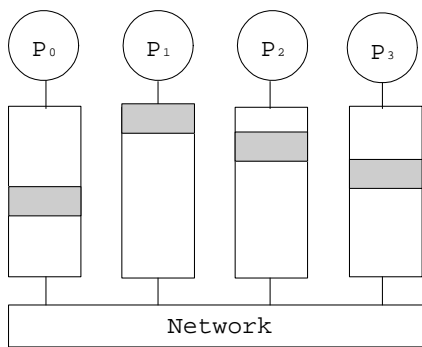
`DsmAllocBlockAt(size, 0)`에 해당한다. 즉, 프로세스 0번부터 `blocksize` 바이트만큼씩 할당한다.



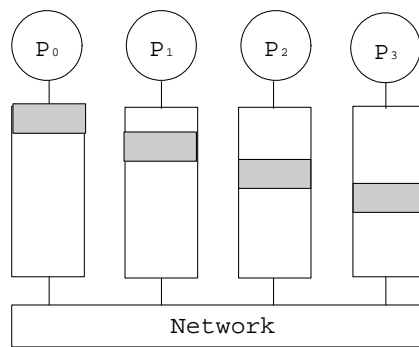
(a) `DsmAllocAt(4MB, 1)`



(b) `DsmAlloc(4MB)`



(c) `DsmAllocBlockAt(4MB, 1MB, 1)`



(d) `DsmAllocBlock(4MB, 1MB)`

그림 3.7: 메모리 할당 함수들의 사용 예

제 4 장

통신 구조

이 장에서는 KDSM의 통신 구조에 대해 설명한다.

KDSM의 통신 모듈은 TCP/IP 상에서 구현되었다. 모든 메시지는 공통의 헤더와 데이터부분으로 구성되어 있다. 메시지를 보내는 것은 `Send(TMsg *msg)` 함수를 호출함으로서 이루어지는데 `msg` 데이터에 기록되어 있는 목적 프로세스의 번호를 보고 원하는 프로세스로 메시지를 전송한다. 받는쪽에서는 특정 파일 디스크립터에 메시지가 왔을때 운영체제가 생성해주는 SIGIO 시그널을 핸들링하여 비동기적으로 메시지를 읽는 방식을 취한다.

1 메시지 구조

모든 메시지는 그림 4.1 과 같이 정의되어 있으며 각 필드의 의미는 다음과 같다.

- *type*
메시지의 타입을 가리킨다.
- *src*
메시지를 전송하는 프로세스의 번호

- *dest*

메시지를 받을 프로세스의 번호

- *size*

data 필드에 들어갈 내용의 크기

- *data*

메시지의 실제 내용. *data* 필드의 내용은 메시지의 type에 따라 다르다.

```
typedef struct {
    int type;
    int src;
    int dest;
    int size;
    char data[MSG_MAXSIZE];
} TMsg;
```

그림 4.1: 메시지 포맷

HLRC와 ScC 의 메시지 타입과 포맷은 부록 A 에 정의되어 있다.

2 프로세스간 통신

2.1 커넥션 생성

TCP는 소켓간의 커넥션을 기본으로 하는 프로토콜이기 때문에 통신 모듈은 초기화 단계에서 각 프로세스간의 커넥션을 생성한다. 각 프로세스는 다른 프로세스에 대한 서버 역할을 하는 동시에 다른 프로세스에게 서비스를 요청하는 클라이언트의 역할 두가지를 모두 해야한다. 따라서 프로세스간의 커넥션은 그림 4.2 와 같은 컴플리트 그래프(complete graph)의 형태를 띈다.

이와같은 커넥션을 구성하기 위해 통신모듈의 초기화 함수인 InitComm() 의 알고리즘은 4.3 과 같다. 각 프로세스는 우선 다른 프로세스로부터의 메시지나 커넥션 요청을 처리할 하나의 서버 소켓을 생성한다. 이 소켓의 포트번호는 하나의 노드에 여러개의 프로세스가 수행 될 수 있으므

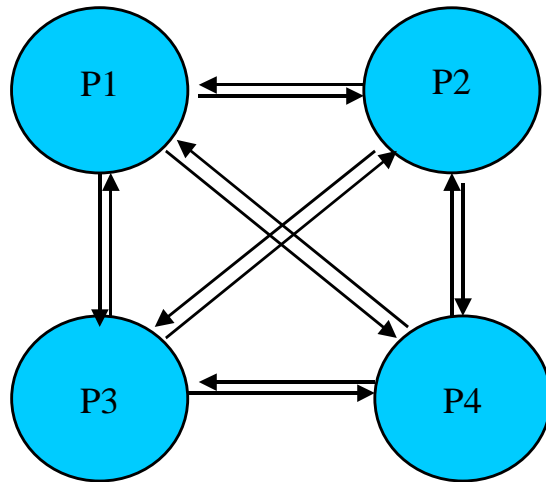


그림 4.2: 프로세스간의 커넥션

로 각 프로세스마다 유일한 값으로 할당하여야 한다. 웹서버와 같은 일반적인 서버 프로그램 달리 KDSM 시스템에서는 소켓에 대해 메시지가 오기를 루프를 돌며 기다리는 것이 아니라 평상시에는 계산을 수행하다가 메시지가 들어왔을 때 이를 비동기적으로 처리하여야 한다. 이와같은 비동기적인 처리를 위해서 네트워크를 통해 새로운 메시지가 도착했을 때 발생하는 시그널인 SIGIO 시그널을 핸들링한다. 핸들러를 등록한 후 각 프로세스는 자기 자신을 제외한 다른 프로세스에게 커넥션 요청을 한다. 이때 커넥션 요청을 받은 프로세스는 SIGIO 시그널 핸들러가 등록되어 있는 상태여야 한다. 커넥션 요청으로 인해 SIGIO 핸들러가 수행되면 핸들러는 이 요청을 받아들이고 이에 해당하는 파일 디스크립터를 할당하게 된다. 모든 프로세스가 이와같은 커넥션을 맺게 되면 앞에서와 같은 컴플리트 그래프 형태의 커넥션이 완성되고 통신모듈의 초기화가 끝나게 된다.

Input: none
Output: none

```

create server socket;
install SIGIO signal handler;
for ( process p in all processes except itself )
{
  connect to p ;
}

```

그림 4.3: InitComm() 알고리즘

2.2 메시지의 전송

메시지의 전송은 메시지를 생성한 후 `Send(TMsg* msg)` 함수에 메시지의 포인터를 넘겨줌으로써 이루어진다. `Send()` 함수는 `msg->dest` 필드의 기록되어 있는 호스트로 메시지를 전송한다. 그림 4.4 는 `Send()` 함수의 알고리즘이다.

Input: TMsg* msg (pointer of message)

Output: none

Algorithm:

```
BeginCS();
if ( gDsmPid == msg->dest )
    MsgServer(msg);
else
    send to msg->dest;
EndCS();
```

그림 4.4: `Send()` 의 알고리즘

`BeginCS()`, `EndCS()` 는 각각 SIGIO 시그널을 disable, enable 하는 역할을 한다. 하지만 이것은 무조건적인 것은 아니며 `BeginCS()` 가 호출된 시점에서 이미 SIGIO 시그널이 disable 되어 있다면 `EndCS()` 가 호출되더라도 시그널을 enable 하지 않는다. 이처럼 메시지 전송중에 SIGIO 시그널이 발생하지 못하도록 하는 것은 메시지의 순차적인 전송을 위한 것이다. 예를들어 메시지의 전송중에 외부에서 메시지가 전송되어 메시지를 처리하기 위한 `MsgServer()` 가 호출되었다고 할 때(`MsgServer` 함수는 메시지 별로 적절한 처리함수를 호출하는 함수로 다음절에서 자세히 설명하도록 하겠다.) `MsgServer()` 가 또다른 메시지를 전송하려 할 경우 문제가 된다. 따라서 `Send()` 시에 SIGIO 시그널이 일어나지 못하도록 함으로서 이와같은 문제가 발생하지 않도록 한다. 메시지의 전송은 자기 자신에게 보내는 경우와 다른 프로세스에게 보내는 경우로 나누어 생각할 수 있다. 목적 프로세스를 나타내는 `msg->dest` 필드와 자신의 프로세스 번호를 가리키는 `gDsmPid` 를 비교하여 자기 자신에게 보내는 경우는 소켓을 통하지 않고 직접 `MsgServer()`를 호출함으로서 필요 없는 시스템 호출과 메모리 복사를 막는다. 목적지가 다른 프로세스일 경우 해당 소켓을 통해 메시지를 전송한다.

2.3 SIGIO 시그널 핸들러

SIGIO 시그널은 새로운 커넥션 요청이 있을때와 전송받은 메시지가 있을경우에 발생한다. 따라서 SIGIO 시그널 핸들러는 이 두 경우에 대한 적절한 처리를 해 주어야 한다. 4.5 는 이 시그널 핸들러의 알고리즘을 기술하고 있다.

Input: none
Output: none
Algorithm:

```
disable SIGIO signal
while ( there's something to be served )
{
    if( there's connection request )
        accept connection

    for ( all connection )
        if ( there's message to be read)
        {
            read(msg);
            MsgServer(msg);
        }
}
enable SIGIO signal
```

그림 4.5: SIGIO 시그널 핸들러 알고리즘

앞절에서 기술한 Send()에서와 같이 시그널 핸들러가 호출되면 가장먼저 SIGIO 시그널을 disable 시키는데 이렇게 하는 것은 메시지의 순차적인 처리를 위해서이다. SIGIO 시그널 핸들러는 SIGIO 시그널이 enable되어 있을때만 호출되므로 앞절의 Send() 에서와 같이 BeginCS(), EndCS()로 현재의 시그널 허용 상태를 체크할 필요가 없다. 새로운 커넥션 요청이 있는 경우 핸들러는 이것을 받아들여 해당 프로세스와의 통로를 설정한다. 읽어들일 메시지가 있을 경우 핸들러는 해당 소켓으로부터 메시지를 읽어들여 *msg* 에 복사한후 MsgServer(*msg*) 를 호출한다. 메시지에 대한 처리가 끝나고 더이상 처리할 것이 없으면 다시 SIGIO 시그널을 enable 하고 마친다.

MsgServer() 함수는 *msg*의 type 필드를 보고 타입별로 적절한 처리 루틴을 호출하는 일종

의 메타 함수이다. 그림 4.6 는 MsgServer() 함수의 알고리즘을 보여준다. 여기에서 ACQ, ACQ-GRANT, BARR 는 각각 acquire 요청, acquire grant 회신, barrier 요청 메시지에 해당하는 것으로 MsgServer()는 이와같이 도착한 메시지의 type 필드를 비교하여 각각의 경우에 해당하는 처리 루틴을 호출한다.

```
Input: TMsg* msg
Output: none
Algorithm:
  switch ( msg->type )
  {
    case ACQ:
      AcqServer(msg);
      break;
    case ACQGRANT:
      AcqGrantServer(msg);
      break;
    case BARR:
      BarrServer(msg);
      break;
    case ..
      .
      .
      .
  }
```

그림 4.6: MsgServer() 알고리즘

제 5 장

Home-based Lazy Release

Consistency의 구현

이 장에서는 HLRC 프로토콜 구현에 대해 설명한다.

1 Home-based Lazy Release Consistency

메모리 일관성 모델(memory consistency model)은 공유메모리 다중처리기에서 아주 중요한 부분이고, 현재까지 많은 모델들이 제안되었다. 일관성 모델은 하드웨어와 프로그래머 간의 계약이라고 볼 수 있다. 즉, 일관성 모델은 프로그램의 연산들이 어떻게 실행되는지 지정하고, 공유 메모리에 대한 접근이 어떤 순서로 일어나는지를 지정한다.

본 장에서 기술할 HLRC 메모리 일관성 모델은 기본적으로 RC(Release Consistency)[2]의 한 구현이라고 할 수 있다. RC는 성능 향상을 위해서 동기화 접근을 acquire와 release 두 연산으로 나누어 생각한다. acquire는 공유된 영역을 접근하는 권한을 얻기 위해서 사용되고, release는 그 권한을 반납한다. RC는 다음과 같이 메모리 접근 순서에 제약을 가한다: (1) 동기화 접근은 서로 sequentially consistent하다. (2) 일반적인 메모리 접근이 허용되기 전에 모든 이전 acquire 연산이 실행되어야 한다. (3) release 연산이 실행되기 이전의 모든 일반적인 메모리 연산이 실행되어야 한

다.

본래의 RC에서는 이와같은 공유메모리 접근 순서에 대한 제약을 지키기 위하여 release시에 변경사항을 전역적으로 수행하도록 구현하였다. 이것을 좀더 개선하여 LRC [10]에서는 다음 acquire를 수행하는 프로세스만이 이러한 변경사항을 수행하도록 하여 전역적 수행으로 인한 병목현상을 해결하였다. HLRC는 LRC와 유사하나 메모리 페이지에 지정된 홈 프로세스가 할당되어 있다는 점이 다르다. HLRC에서는 release시 수정된 페이지에 대한 정보를 홈 프로세스에 적용시키며 나중에 다른 프로세스가 해당 메모리 페이지를 액세스할때 할 때 홈 프로세스에게서 해당 페이지 전체를 받아온다.

HLRC는 DSM 응용 프로그램의 수행을 interval들로 구분하는데 한 interval은 release와 다음 release 사이를 말한다. Interval은 release 마다 증가하며 interval 내에서의 모든 변경된 페이지에 대한 정보는 특정한 테이블에 기록한다. HLRC에서 여러 프로세스간의 메모리 일관성은 interval을 기본 단위로 하여 이루어진다.

일관성의 유지를 위해 각 프로세스는 각각 독립적으로 벡터 타임스탬프(vector timestamp)를 유지한다. 벡터 타임스탬프는 어떤 프로세스가 알고 있는 다른 프로세스의 최신의 interval 값의 리스트이며 이 벡터 값을 통해 어떤 프로세스가 다른 프로세스의 어느 시간까지의 수정된 값을 가지고 있는지를 알 수 있다. 예를 들어 P_0, P_1, P_2 의 세 개의 프로세스로 이루어진 환경에서 P_1 의 벡터가 (1,3,2)라고 하면 이는 그때까지 P_1 에 입장에서 P_0 는 interval 1까지, P_1 은 3까지, P_2 는 2까지의 수정된 사항을 알고 있음을 뜻한다. 벡터의 수정은 자기 자신의 인덱스의 경우 lock을 release할때 1씩 증가하며 다른 프로세스의 인덱스는 lock grant 메시지를 받을 때 이전 lock 소유자의 벡터 타임스탬프와 자신의 것을 비교해 봄으로서 수정한다.

각 프로세스는 벡터 타임스탬프와 함께 interval 내의 변경사항에 대한 정보, 즉 write-notice를 기록하는 테이블을 유지하는데 이 테이블을 통해 특정 프로세스의 특정 interval동안의 변경사항을 해당 프로세스에게 직접 요청하지 않아도 알 수 있다. 이 write-notice는 lock을 acquire할때 이전에 lock을 가지고 있던 프로세스가 grant 메시지에 포함시켜서 전송해준다.

2 Lock의 구현

일반적으로 lock은 상호배제를 위한 도구이나 HLRC에서는 상호배제 뿐만 아니라 lock을 통해 메모리 일관성에 대한 정보를 교환한다. 이 절에서는 KDSM 시스템의 lock 구현에 대해 상호배제와 일관성 유지의 순서로 기술하도록 한다.

2.1 상호배제 (Mutual Exclusion)

상호배제는 토큰을 기본으로 하는 분산 lock 모델을 사용하였다. 이 방법에서 lock을 acquire하기 위해서는 lock 토큰을 받아야 하며 이것은 바로 전에 lock을 요청했던 프로세스에게서 받아오게 된다. 예를 들어 현재 어떤 프로세스가 n번째로 lock을 요청했다면 n-1번째 lock을 요청한 프로세스로부터 lock 토큰을 받아오게 된다. 여기에서 문제는 n-1번째 lock을 요청한 프로세스가 어떤 것인지 어떻게 알 수 있는가하는 것인데, 여기에서는 지정된 manager를 두어 이 정보를 유지하게 하였다. 때문에 lock 토큰을 가지고 있지 않을 경우 lock을 획득하기 위해서는 우선 manager에게 요청을 한 후, manager에 의해 이전 lock 요청자에게 이것이 포워딩 되고, 다시 grant메시지를 원래의 요청자에게 보내는 3번의 메시지 전송이 필요하다.

이와 같은 처리를 위해서 각 프로세스는 다음의 세 필드(field)의 값을 유지한다.

- *local*

lock 토큰을 가지고 있는지의 여부

- *hold*

lock 토큰을 사용하고 있는지의 여부

- *saved*

아직 처리하지 못한 포워딩된 lock 요청 메시지를 저장한다.

그림 5.1은 이 필드들이 어떻게 관리되는지를 보여주고 있다. (a)는 최초의 상태로서 lock 토큰이 P_1 에 있으며 lock manager는 *next*를 P_1 으로 하여 P_1 에게 lock 토큰이 있음을 가리키고 있

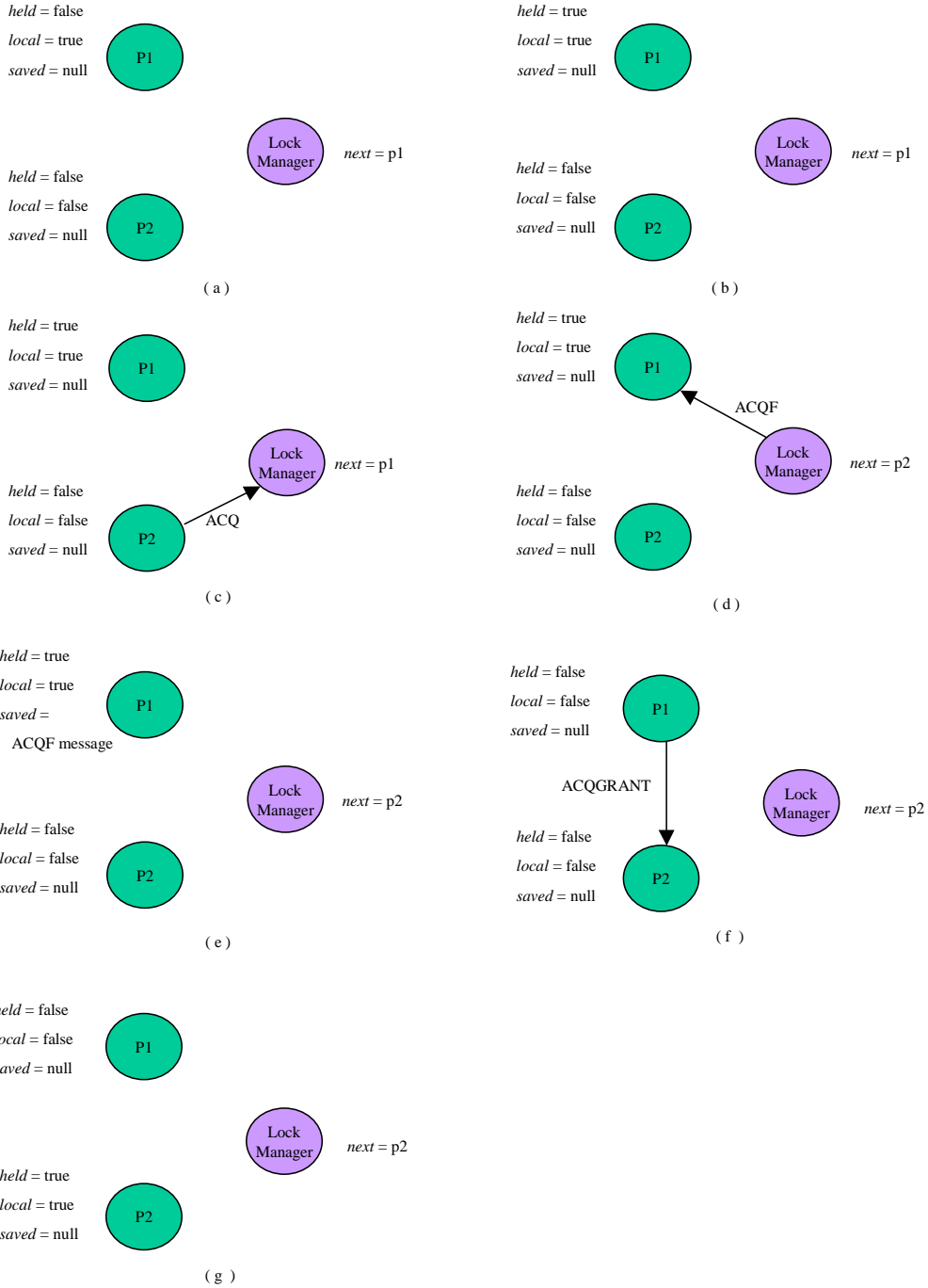


그림 5.1: 분산 Lock 관리

다. (b)는 P_1 이 acquire를 했을 경우인데 *local* 이 true, 즉 lock 토큰이 자신에게 있으므로 lock manager를 거치지 않고 *held*를 true로 함으로서 바로 lock을 획득하였다. (c)에서 P_2 가 lock을 획득하려 할 때 *local* 이 false 이므로 lock manager에게 ACQ 메시지를 보내 lock을 요청한다. (d) lock manager는 *next* 를 보고 P_2 의 ACQ 메시지를 P_1 에게 포워딩시키고 *next* = P_2 로 설정한다. 이때 lock manager는 이 메시지가 포워딩된 것임을 알려주기 위해 메시지의 type을 ACQF 로 바꾸어 전송한다. ACQF 메시지는 ACQ메시지의 *type* 필드만을 ACQF로 바꾼 메시지이다. (e) 포워딩된 lock 요청 메시지를 받은 P_1 은 그러나 아직 lock을 release하지 않았기 때문에 바로 grant할 수 없으므로 이를 *saved* 필드에 저장한다. (f) P_1 이 lock을 release 할때 *saved* 필드를 체크하는데 lock manager를 통해 포워딩된 P_2 의 ACQF 메시지가 저장되어 있으므로 ACQGRANT메시지를 생성하여 P_2 에게 lock을 grant해 주게 된다.

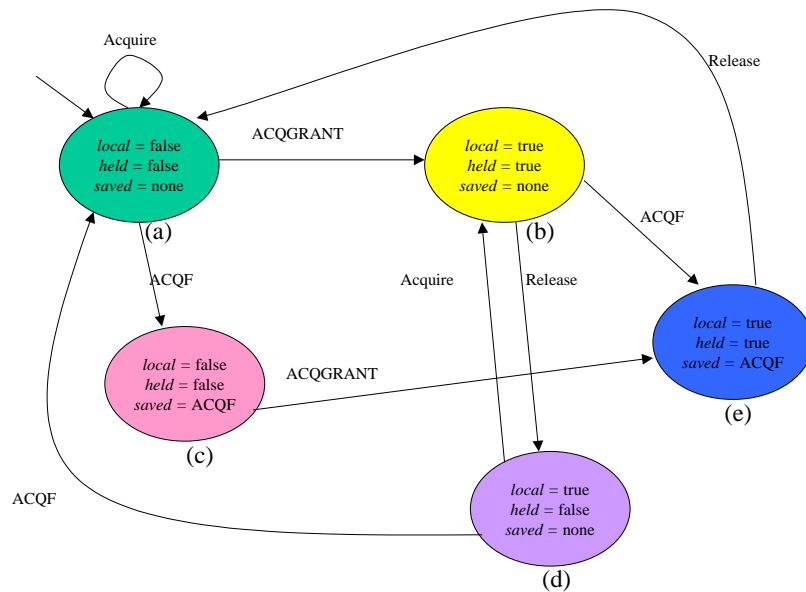


그림 5.2: Lock 상태

그림 5.2 은 한 프로세스에서 lock의 상태 전이도이다. lock은 그림에서와 같이 *local* , *held*, *saved* 세 변수에 따른 5개의 state 와 9개의 상태변화를 가진다. 상태변화가 일어나는 것은 그 프로세스가 lock을 acquire 혹은 release 하거나 외부에서 ACQF (acquire forwarded), ACQGRANT(acquire grant) 메시지를 받는 경우이다.

(a)는 lock을 획득하지 않았으며 ($held == false$), lock토큰을 가지고 있지 않고($local == false$) 포워딩된 메시지가 없는($saved == null$) 상태를 나타낸다. 이 상태에서 프로세스는 acquire 연산을 수행할 수 있으며 ACQGRANT 메시지나 포워딩된 lock요청인 ACQF 메시지를 받으면 상태변화가 일어난다. (b)는 lock을 획득한 상태로 release를 하거나 ACQF 메시지를 받은 경우 상태변화가 일어나게 된다. (c)는 (a) 상태에서 ACQF 메시지를 받은 경우이다. 이 상태에서 프로세스는 ACQGRANT 메시지가 오기를 기다리며 ACQGRANT 메시지를 받으면 (e) 상태로 바뀐다. (d)는 (b)에서 release를 수행하였을 경우에 생기는 상태로 lock 토큰을 가지고 있으나 이를 사용하고 있지 않은 상태이다. lock토큰을 가지고 있기 때문에 acquire를 수행하면 (b)상태로 상태가 바뀌고 바로 lock을 쓸 수 있다. 만일 이 상태에서 ACQF 메시지를 받게 되면 메시지를 생성한 프로세스에게 ACQGRANT 메시지를 보내주어 lock토큰을 넘겨주게 된다. (e) 상태는 (b)와 마찬가지로 lock을 획득한 상태이나 포워딩된 요청이 있기 때문에 release를 하게 되면 ACQGRANT 메시지를 보내야 한다.

2.2 메모리 일관성(Consistency)

HLRC 프로토콜에서 lock의 가장 중요한 기능은 바로 메모리 일관성 정보를 주고받는 것이다. 메모리 일관성 정보는 앞서 언급한 write-notice 정보인데 이 정보는 lock을 release하는 프로세스에서 lock을 acquire하는 프로세스에게 전달된다. 앞서 각 프로세스는 각각 벡터 타임 스탬프를 가지고 있다고 하였다. 예를들어 어떤 프로세스의 벡터 타임 스탬프 값이 (1, 3, 2) 라고 한다면 이 프로세스는 P_0 의 interval 1, P_1 의 interval 3, P_2 의 interval 2까지의 변화를 알고 있다는 의미이다. 이것은 다시말해 그때까지의 write-notice 정보를 가지고 있다는 의미이다. 때문에 각 프로세스는 모든 프로세스에 대해 interval 별로 write-notice를 저장하는 테이블을 유지하여야 한다. 이 테이블에는 프로세스의 벡터 타임스탬프에 기록된 각각의 프로세스의 interval까지의 write-notice 정보가 기록된다. 테이블의 내용이 다른 프로세스에게로 전달되는 것은 lock grant 메시지를 전송할 때이다.

프로세스가 lock을 얻기 위해서는 ACQ 메시지를 보내야 하는데 이때 자신의 벡터 타임스탬프를 ACQ메시지에 포함시킨다. 이 ACQ메시지는 lock manager를 거쳐 메시지의 *type* 필드만 ACQF 로 바뀌어 lock을 release하는 프로세스에게로 전송된다. release하는 프로세스는 ACQF에 기록된 벡터 타임스탬프와 자신의 벡터 타임스탬프를 비교함으로써 lock을 요청한 프로세스가 모르고 있는 interval들에 대한 정보를 알 수 있다. release하는 프로세스는 이들 interval에 해당하는 write-notice 정보를 ACQGRANT메시지에 포함시켜 lock을 요청한 프로세스에게로 전송한다. ACQGRANT 메시지를 받은 프로세스는 이 write-notice 정보를 통해 자신의 memory를 적절히 invalid시키고 자신의 벡터 타임스탬프를 갱신하며 또한 자신의 테이블에 이 write-notice정보를 기록한다.

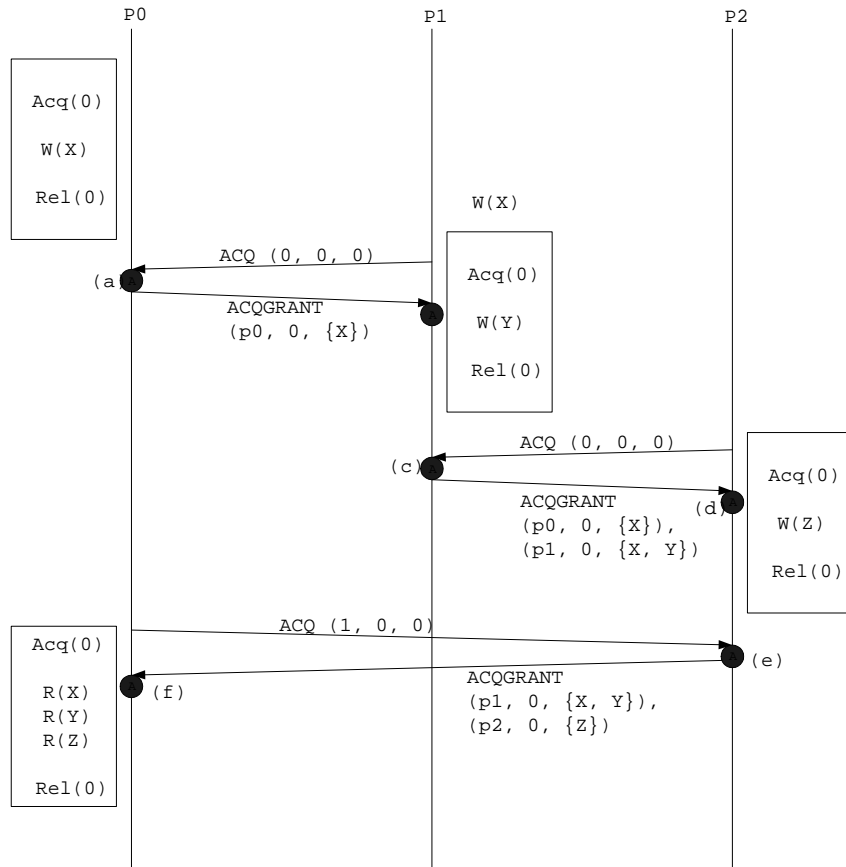


그림 5.3: 메모리 일관성 정보의 흐름

그림 5.3 은 HLRC 프로토콜에서의 메모리 일관성 정보의 흐름의 예를 보여준다. 그림 5.4 는 그림 5.3에서 ACQGRANT 메시지를 전송할때와 ACQGRANT메시지를 받을 때 해당 프로세스

(a) 에서의 P0의 상태

- 벡터 타임스탬프 : (1, 0, 0)

- write-notice 테이블

	0	1	2
P0	X		
P1			
P2			

(b) 에서의 P1의 상태

- 벡터 타임스탬프 : (1, 0, 0)

- write-notice 테이블

	0	1	2
P0	X		
P1			
P2			

(c) 에서의 P1의 상태

- 벡터 타임스탬프 : (1, 1, 0)

- write-notice 테이블

	0	1	2
P0	X		
P1	X,Y		
P2			

(d) 에서의 P2의 상태

- 벡터 타임스탬프 : (1, 1, 0)

- write-notice 테이블

	0	1	2
P0	X		
P1	X,Y		
P2			

(e) 에서의 P2의 상태

- 벡터 타임스탬프 : (1, 1, 1)

- write-notice 테이블

	0	1	2
P0	X		
P1	X,Y		
P2	Z		

(f) 에서의 P0의 상태

- 벡터 타임스탬프 : (1, 1, 1)

- write-notice 테이블

	0	1	2
P0	X		
P1	X,Y		
P2	Z		

그림 5.4: 벡터 타임스탬프와 write-notice 정보

의 벡터 타임스탬프와 write-notice를 저장하는 테이블의 내용을 나타내고 있다. 이 그림의 시나리오의 다음과 같다. : P_0 는 lock을 얻은 후 페이지 X를 변경한 후 lock을 반환한다. 다음으로 P_1 이 lock을 얻고 페이지 Y를 수정한다. P_1 은 lock을 얻기 이전에 페이지 X를 변경하였다. 다음으로 P_2 가 lock을 얻고 페이지 Z를 변경한다. 마지막으로 P_0 가 다시 lock을 얻고 X, Y, Z를 읽는다.

그림에서 처음에 P_0 가 lock 토큰을 가지고 있다고 가정한다. 또한 본래 ACQ 메시지는 lock manager를 통해 ACQF 메시지로 바뀌어 포워딩 되지만 이는 메모리 일관성 정보의 흐름과는 무관하며 앞절에서 상세히 설명하였으므로 여기에서는 생략하였다. 따라서 ACQ 메시지가 바로 이전 lock 요청을 했던 프로세스에게로 전달되는 것으로 표시하였다. ACQ 메시지의 괄호안의 숫자들은 자신의 벡터 타임스탬프를 나타낸다. ACQGRANT 메시지는 write-notice 정보를 포함하는데 write-notice는 (프로세스, interval, 수정된 페이지 리스트) 와 같은 형태로 나타내었다. 즉 $(P_0, 1, \{X, Y\})$ 는 P_0 프로세스가 interval 1에 X, Y 페이지를 수정하였음을 뜻한다. 최초에 모든 프로세스의 벡터 타임스탬프는 $(0,0,0)$ 으로 초기화 되어 있으며 write-notice 테이블은 모두 비어있다.

(a) P_0 가 lock을 이용한 작업을 끝내고 ACQGRANT 메시지를 받았을 경우인 를 살펴보자. 벡터 타임스탬프에서 자신의 벡터 값이 증가하는 시점은 lock을 release할 때이다. P_0 는 (a) 이전에 lock을 release 한 상태이기 때문에 P_0 의 벡터 타임스탬프는 $(1,0,0)$ 이다. 이때의 P_0 의 상태가 그림 5.4 의 (a) 이다. P_1 으로부터 전송되어 온 ACQ 메시지에서 P_1 의 벡터 타임스탬프는 $(0,0,0)$ 임을 알 수 있는데 이 시점에서 P_0 는 P_1 이 P_0 의 interval 0에 대한 변화를 알지 못함을 알 수 있다. 따라서 P_0 는 ACQGRANT 메시지에 write-notice $(P_0, 0, \{X\})$ 를 포함시켜 P_1 에게 전송한다.

(b) P_1 이 ACQGRANT 메시지를 받았을때를 살펴보자. ACQGRANT 메시지에 포함된 write-notice $(P_0, 0, \{X\})$ 를 통해 P_1 은 P_0 가 interval 0에 페이지 X에 대해 변경하였음을 알 수 있다. 따라서 P_1 은 페이지 X를 무효화 시키고 P_0 의 현재 interval이 1 이상임을 알 수 있으므로 자신의 벡터 타임스탬프를 $(0,1,0)$ 으로 갱신한다. 또한 $(P_0, 0, X)$ 를 자신의 테이블에 저장해 다른 프로세스에게 P_0 의 write-notice를 알려줄 수 있도록 한다. 이와같이 갱신한 후의 P_1 의 상태가 그림 5.4 의 (b) 이다.

(c) P_1 이 P_2 로부터 ACQ 메시지를 받았을 때를 살펴보자. P_1 은 이에 앞서 lock을 release 하

였으므로 현재 P_1 의 벡터 타임스탬프는 $(1,1,0)$ 이다. P_2 의 타임스탬프 값은 $(0,0,0)$ 이므로 P_1 은 write-notice $(P_0, 0, \{X\})$ 와 $(P_1, 0, \{X, Y\})$ 를 ACQGRANT 메시지에 포함시켜 전송한다. 그림 5.4의 (c)는 P_2 로부터 ACQ 메시지를 받았을 때의 P_1 의 상태이다.

(d) P_1 으로부터 받은 ACQGRANT 메시지를 통해 P_2 는 자신의 벡터 타임스탬프를 $(1,1,0)$ 으로 갱신하고 페이지 X, Y를 무효화 시킨다. 그림 5.4의 (d)는 이와같이 갱신한 후의 P_2 의 상태이다.

(e) P_0 로부터 ACQ 메시지 $(1,0,0)$ 을 받았을 때 P_2 의 벡터 타임스탬프는 $(1,1,1)$ 이다. 따라서 P_1 과 P_2 의 interval 0에 대한 write-notice인 $(P_1, 0, \{X, Y\})$, $(P_2, 0, \{Z\})$ 를 ACQGRANT에 포함시켜 P_0 에 전송한다. 그림 5.4의 (e)는 P_0 로부터 ACQ 메시지를 받았을 때의 P_2 의 상태이다.

(f) P_0 가 다시 lock을 요청하여 ACQGRANT 메시지를 받았을 때를 살펴보자. 이때 P_0 의 벡터 타임스탬프는 $(1,0,0)$ 인데 ACQGRANT 메시지에 포함된 write-notice $(P_1, 0, \{X, Y\})$, $(P_2, 0, \{Z\})$ 를 통해 P_1 과 P_2 의 벡터가 1 이상임을 알 수 있다. 따라서 자신의 벡터 타임스탬프를 $(1,1,1)$ 로 수정하고 X, Y, Z 페이지를 무효화 시킨다. 그림 5.4의 (f)는 이와같이 갱신한 후의 P_0 의 상태이다.

2.3 알고리즘

앞에서 기술한 lock을 구현하는데에는 사용자에게 보여지는 프로그램 인터페이스인 acquire, release 명령과 ACQ 메시지를 처리하는 AcqServer(), 포워딩된 ACQ 메시지를 처리하는 AcqForwardServer(), 그리고 ACQGRANT 메시지를 처리하는 AcqGrantServer가 구현되어야 한다. 다음은 이들 각각에 대한 알고리즘이다.

그림 5.5는 lock acquire를 위한 DsmLock() 프로시저의 알고리즘이다. lock 토큰을 가지고 있는 경우 즉 local이 true 일 경우에는 held를 true로 바꾸어주고 바로 return한다. lock 토큰을 가지고 있지 않은 경우는 lock manager에게 ACQ 메시지를 보낸다. ACQ 메시지에는 자신의 벡터 타임스탬프(vector)를 포함시킨다. ACQGRANT 메시지가 도착할 때까지 대기한 수, 메시지가 도착하면 함께 온 write notice 정보(wtns)에 따라 해당 캐쉬 페이지들을 무효화 시키고 벡터 타

임스탬프를 갱신한다. *local*과 *held* 필드는 ACQGRANT 메시지를 받는 AcqGrantServer() 함수가 세팅해 준다. 그림 5.6 는 AcqGrantServer()의 알고리즘이다.

Input: lockid
 Output: none
 Algorithm:

```

if ( local == true )
    held = true;
else
{
    send diff for all dirty pages;
    send ACQ(vector) to lock manager;
    wait until ACQGRANT(wtns);
    invalidate cache pages according to wtns;
    add wtns to the write-notice table;
    update vector according to wtns;
}

```

그림 5.5: DsmLock() 알고리즘

```

local = true;
held = true;

```

그림 5.6: AcqGrantServer() 알고리즘

그림 5.7 는 lock release 를 위한 DsmUnlock() 프로시저의 알고리즘이다. release 시에는 먼저 현재 interval 내에서의 모든 변경된 페이지들의 diff를 생성하여 이를 홈 노드에 적용시킨 후 새로운 interval을 생성한다. 만일 포워딩된 ACQF 메시지가 저장되어 있을 경우 ACQF 메시지의 *src* 필드를 보고 그 프로세스에게 ACQGRANT(*wtns*) 메시지를 보낸다. *wtns* 는 write-notice를 뜻하며 이것은 ACQF 메시지에 포함되어 있는 벡터 타임스탬프와 lock 을 release하는 프로세스의 벡터 타임스탬프를 비교하여 생성한다.

그림 6.7 은 lock manager 프로세스에서 ACQ메시지가 왔을 때 불리는 AcqServer() 함수이다. ACQ메시지는 일차적으로 lock manager에게 전송된 후, 이곳에서 유지되는 *next* 필드에 기록되어 있는 프로세스에게 포워딩된다. 이때 메시지의 타입(*msg->type*)을 ACQF 로 바꾸어 줌으로서

Input: lockid
Output: none
Algorithm:

```
send DIFF for all dirty page;  
create new interval;  
if ( saved ACQF message )  
{  
    compare vector timestamp in ACQF and mine  
    then make wtns;  
    send ACQGRANT(wtns);  
}
```

그림 5.7: DsmUnlock() 알고리즘

이것이 포워딩된 메시지임을 알수 있게 한다. 이와같이 함으로서 lock 요청은 전체적으로 분산 선입선출 큐(distributed FIFO queue)로서 유지된다.

Input: TMsg* *msg* (ACQ message)
Output: none
Algorithm:

```
msg->type = ACQF  
send msg to next  
next = msg->src
```

그림 5.8: AcqServer() 알고리즘

그림 5.9 은 lock manager에 의해 포워딩된 ACQF 메시지를 처리하는 AcqForwardServer() 함수의 알고리즘이다. lock 토큰을 가지고 있으나 이를 사용하지 않는 경우(*local* == true && *held* == false) 엔 *wtns* 를 생성하여 바로 ACQGRANT 메시지를 전송하고, 나머지 경우엔 메시지를 저장해서 추후 lock release시에 ACQGRANT 메시지를 보낼 수 있도록 한다.

Input: TMsg* *msg* (ACQF message)

Output: none

Algorithm:

```
if ( local == true )
{
  if ( held == true )
    saved = msg;
  else
  {
    compare vector timestamp in ACQF and mine
    then make wtns;
    send ACQGRANT(wtns);
    local = false;
  }
}
else
  saved = msg;
```

그림 5.9: AcqForwardServer() 알고리즘

3 Barrier의 구현

barrier는 release와 acquire로 생각할 수 있다. 즉 barrier에 도착하는 것은 lock release, barrier를 떠나는 것은 lock acquire로 생각할 수 있다. barrier 도착 시 고정된 barrier 서버에게 자신의 벡터 타임스탬프와 write-notices를 보내면 barrier 서버는 모든 프로세스에게 이를 종합하여 각 requester의 벡터 타임스탬프와 비교하여 write-notices를 보내준다.

그림 5.10 는 DsmBarrier() 함수의 알고리즘을 보여준다. DsmBarrier() 가 barrier server에서 호출되었다면 *barrier_counter*를 증가시킨 후 이 값이 전체 프로세스의 갯수와 동일할때까지 대기한다. 모든 프로세스에게서 BARR 메시지를 받게 되면 각 프로세스가 BARR 메시지에 포함시켜 보내온 벡터 타임스탬프(*vector_in*)와 write-notice를 취합한 *wtns_all* 을 이용하여 *wtns*를 생성하고 이를 전송한다.

barrier server가 아닌 프로세스에게서 호출되었을 경우 BARR 메시지에 자신의 벡터 타임스탬프(*vector*)와 write-notice를 포함시켜 barrier server에게 전송하고 BARRGRANT를 받을때까

지 대기한다. BARRGRANT 메시지를 받으면 함께 온 write-notice 정보(*wtnnts_in*) 에 따라 해당 캐쉬 페이지들을 무효화 시킨다.

Input: none
Output: none
Algorithm:

```
create new interval;  
  
if ( this is barrier server )  
{  
    increment barrier_counter;  
    wait until (barrier_counter == total number of processes);  
    invalidate all cache page according to received wtnnts_all;  
    for (process p in all processes except barrier server )  
    {  
        make wtnnts according to vector_in(p 's vector timestamp);  
        send BARRGRANT(wtnnts);  
    }  
}  
else  
{  
    send BARR(vector, wtnnts ) message to barrier server;  
    wait until BARRGRANT(wtnnts_in);  
    invalidate all cache page according to wtnnts_in;  
}  
update vector;
```

그림 5.10: DsmBarrier() 알고리즘

그림 6.10 는 BARR 메시지가 불리는 BarrServer() 함수의 알고리즘을 보여준다. 이 서버는 지정된 barrier server 프로세스에서만 호출되며 BARR 메시지가 올때마다 *barrier_count*를 1씩 증가시키고 BARR 메시지에 포함되어 전송된 각 프로세스의 벡터 타임스탬프(*vector_in*)와 *wtnnts*를 기록한다.

Input: TMsg* *msg* (BARR message)

Output: none

Algorithm:

```
save requester's vector timestamp to vector_in ;  
add wtnts in BARR message to wtnts_all;  
increment barrier_counter;
```

그림 5.11: BarrServer() 알고리즘

제 6 장

Scope Consistency 프로토콜의 구현

이 장에서는 scope consistency(ScC)에 관해서 간단하게 정리한 후, ScC를 위한 프로토콜 및 구현 사항에 대해서 설명한다.

1 Scope Consistency [1]

ScC는 데이터와 동기화 사건들 간의 암묵적인 관계를 사용해서 consistency scope를 유추한다. 어떤 scope 내에서 데이터에 가해진 수정 사항은 그 scope 내에서만 보여지는 것이 보장된다. 같은 lock에 의해서 보호되는 임계영역을 하나의 consistency scope로 생각할 수 있고, barrier는 전체 프로그램을 포함하는 전역적인 consistency scope로 생각할 수 있다. 각 consistency scope는 scope를 여는 *open* 연산과 scope를 닫는 *close* 연산을 가진다. 어떤 프로세스가 consistency scope를 열고 있는 기간을 *session*이라 한다. 한 consistency scope session 내에서 행해진 모든 수정 사항은 그 scope의 새로운 session에 들어가는 프로세스에게 보여진다. 하지만, scope session 밖에서 행해진 수정 사항은 보여지는 것이 보장되지 않는다.

2 Lock의 구현

ScC 프로토콜은 페이지 기반의 무효화 프로토콜(page-based invalidation protocol)과 홈 기반의 프로토콜(home-based protocol)을 사용해서 구현된다. 홈 기반의 프로토콜에서, 어떤 페이지에 가해진 수정들은 그 페이지의 홈으로 직접 전달된다. 따라서 홈 노드의 페이지들은 최신 정보를 유지하고, 페이지 폴트가 났을 때 여러 노드가 아닌 홈 노드에서 전체 페이지를 읽어온다.

ScC는 각 lock에 대해서 incarnation 번호를 가진다 [1]. incarnation 번호는 초기에 모든 프로세스에서 0으로 설정되고, lock이 release 될 때마다 1씩 증가한다. incarnation에 관한 정보는 lock마다 고정적으로 할당되는 홈에서 관리된다. 즉, 각 lock의 홈에서는 일종의 lock manager가 있어 자신이 관리하는 lock에 관해서 다음과 같은 정보를 유지한다:

- 현재 incarnation 번호
- 각 incarnation에서 수정된 페이지 리스트, 즉 write notice
- lock 요청을 순서적으로 서비스하기 위한 lock request queue

모든 프로세스는 자신이 얻었던 모든 lock의 마지막 incarnation 번호를 기억한다. 프로세스는 lock을 얻기 위해서 ACQ 메시지를 보낼 때, 해당 lock에 대해서 자신이 마지막으로 생성했던 incarnation 번호를 함께 보낸다. lock manager는 이 정보를 기반으로 ACQ를 요청한 프로세스에게 ACQGRANT 메시지와 함께 현재 incarnation 번호와 write notice를 전송한다. ACQGRANT를 받은 프로세스는 자신의 incarnation 번호를 자신이 기억하는 마지막 incarnation 번호에 1을 더한 값과 ACQGRANT 메시지와 함께 온 incarnation 번호에 1을 더한 값 중 큰 값으로 설정한다. 그리고, write notice에 표시된 페이지들을 무효화시킨다.

2.1 Lock 프로토콜의 실행 예

그림 6.1은 다음과 같은 시나리오를 가진 ScC 프로토콜의 실행 예를 보여준다: 프로세스 P_0 이 lock을 얻어서 X 페이지를 변경한 후 lock을 반환한다. 다음으로, 프로세스 P_1 이 lock을 얻고 Y 페

이지를 변경한 후 lock을 반환한다. 다음으로, 프로세스 P_2 가 lock을 얻고 Z 페이지를 변경한 후 lock을 반환한다. 마지막으로, 프로세스 P_0 이 다시 lock을 얻어서 X, Y, Z를 읽는다.

앞서 얘기했듯이 lock manager를 포함한 모든 프로세스들은 모든 lock에 대해서 incarnation 번호를 0으로 초기화한다. 첫번째, P_0 이 ACQ 메시지를 lock의 홈에 전송한다. 그림에서 ACQ 메시지 괄호 안의 수는 ACQ 메시지를 보내는 프로세스가 기억하는 마지막 incarnation 번호를 표시한다. lock 홈에서는 ACQ 메시지가 들어왔을 때 현재 lock을 가진 프로세스가 없으므로 바로 ACQGRANT 메시지를 보낸다. 이 때 P_0 의 incarnation 번호와 lock manager의 incarnation 번호가 같으므로 write notice를 보낼 것이 없다. ACQGRANT 메시지 괄호 안의 0은 lock manager의 현재 incarnation 번호이고, {}는 write notice를 표시한다.

P_0 은 ACQGRANT를 받으면 lock 0에 대한 incarnation 번호를 1로 설정한다. P_0 은 작업이 끝나면 lock을 반환하기 위해 lock의 홈으로 REL 메시지를 전송한다. 이 때 REL 메시지에 자신의 incarnation에서 수정한 페이지들의 리스트, 즉 write notice를 함께 보낸다. lock의 홈에서는 REL 메시지를 받으면 현재 incarnation 번호를 1 증가시킨 후, 그 incarnation에 관련된 write notice 정보를 저장한다. 즉, A 지점에서 현재 incarnation은 1이 되고, write notice는 {X}이다. 그림 6.2는 lock 홈에서 관리되는 정보를 보여준다.

P_1 이 ACQ 메시지를 보낼 때를 살펴보자. P_1 은 현재 incarnation 번호가 0이고, lock 홈의 incarnation 번호는 1이기 때문에 P_1 은 incarnation 1을 아직 보지 못했다고 볼 수 있다. 따라서 lock manager는 ACQGRANT 메시지에 incarnation 1에서 수정된 페이지 리스트를 함께 보낸다. P_1 은 페이지 리스트에 해당하는 페이지들을 무효화시킨다.

P_2 가 ACQ 메시지를 보낼 때를 살펴보자. P_2 는 현재 incarnation 번호가 0이고, lock 홈의 incarnation 번호는 2이기 때문에, lock manager는 ACQGRANT 메시지에 incarnation 1, 2에서 수정된 페이지 리스트를 함께 보낸다. P_2 는 페이지 리스트에 해당하는 페이지들을 무효화시킨다.

마지막으로 P_0 이 다시 lock을 얻기 위해서 ACQ 메시지를 보낼 때를 보자. P_0 은 현재 incarnation 번호가 1이기 때문에, incarnation 2, 3을 아직 보지 못했다. 따라서, lock manager는 ACQGRANT 메시지에 incarnation 2, 3에서 수정된 페이지인 {Y, Z}를 write notice로써 보낸다. P_0 은

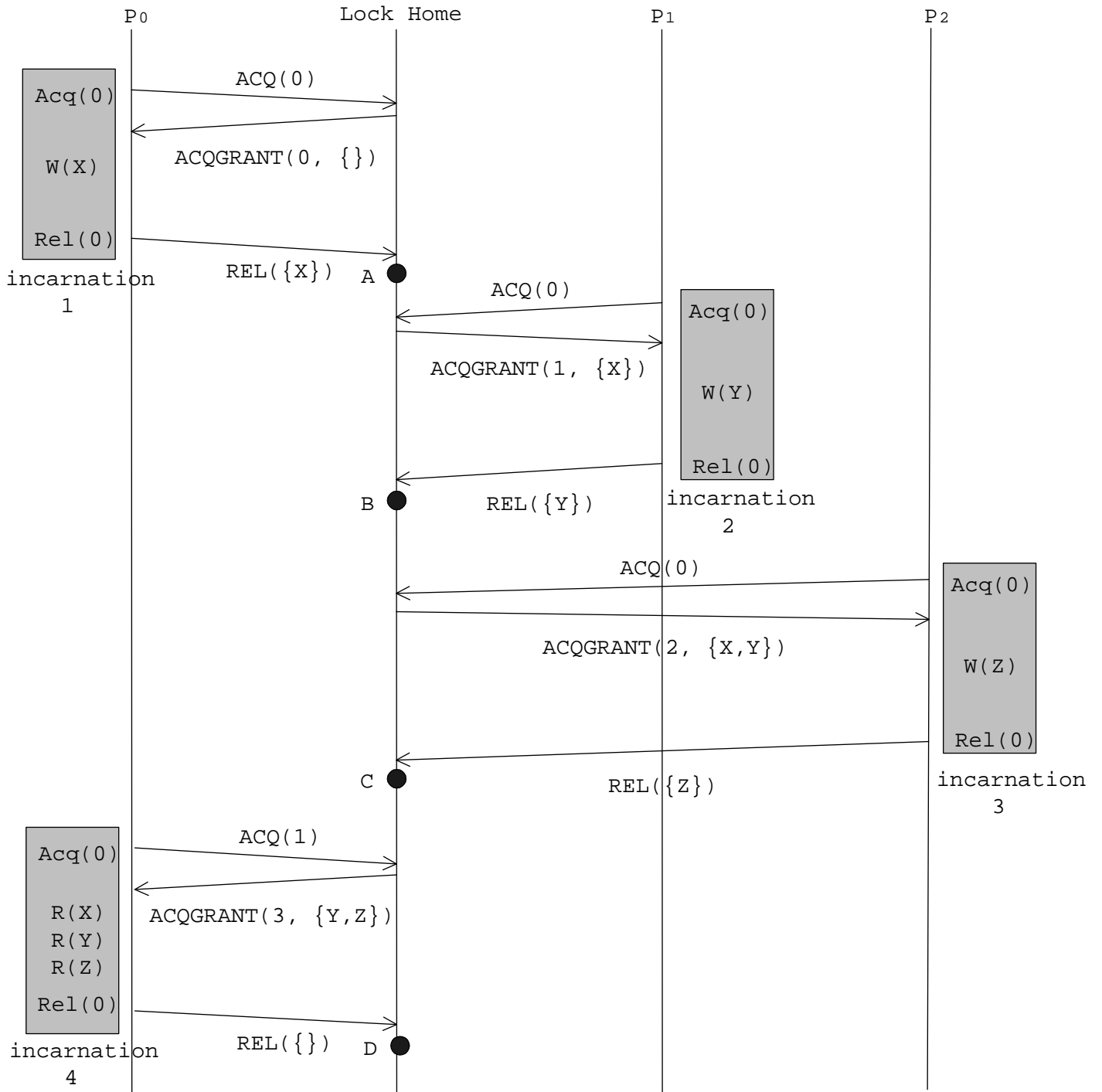


그림 6.1: ScC 프로토콜의 실행 예

incarnation	write notices	point in Fig 6.1
0	-	-
1	X	A
2	Y	B
3	Z	C
4	-	D

그림 6.2: lock 홈에서 유지하는 정보

ACQGRANT를 받고 자신의 새로운 incarnation을 4로 설정하고, write notice에 따라서 페이지 Y, Z를 무효화시킨다. 잇따라 일어나는 X, Y, Z 페이지에 대한 읽기는 페이지 폴트를 발생시켜 각 페이지의 홈에서 전체 페이지를 읽어 와서 계속 진행한다.

2.2 중첩된 lock의 구현

그림 6.3은 lock이 중첩된 프로그램의 예를 보여준다. lock이 중첩되어 사용될 때, 각 lock의 scope 내에서 어떤 페이지들이 수정되었는지 결정하기 위해서 다음과 같은 사항들이 고려되어야 한다:

- 첫째, 내부 임계영역에서 X0이 수정되었는지 결정하기 위해서는 내부 임계영역에 들어가기 전에 외부 임계영역에서 수정된 모든 페이지들을 RO 상태로 초기화를 해야한다.
- 둘째, 내부 임계영역에 들어가기 전에 X0이 외부 임계영역에서 수정되었음을 기록하기 위해서는 X0의 diff와 write notice를 어디가에 저장해 둘 필요가 있다.

```

Acq(0)
  X0 = 1;
  ...
Acq(1)
  X0 = 2;
Rel(1)
  ...
  X0 = 3;
Rel(0)

```

그림 6.3: 중첩된 lock 구조의 예

- 셋째, 내부 임계영역을 벗어난 후 X0이 수정되었는지 결정하기 위해서는 내부 임계영역을 벗어날 때 내부 임계영역에서 수정된 모든 페이지들을 다시 RO 상태로 바꿔야 한다.
- 넷째, 내부 임계영역에서 수정된 페이지들은 외부 임계영역에서도 수정된 페이지로 보아야 한다.

이로부터 알 수 있는 것은 새로운 임계영역에 들어 갈 때와 임계영역에서 벗어날 때마다 RW 상태인 모든 페이지를 RO 상태로 바꾸어야 한다는 것이다. 또한 RW 페이지들이 RO로 변하기 전에 이 페이지들에 대한 diff 및 write notice를 계산해야 하고, 필요에 따라 저장되어야 한다. 이를 위해서 스택 구조를 사용한다.

```

Input: none
Output: wtnt (write notices)
Algorithm:
  for (all RW cache pages)
  {
    add this page to wtnt;
    calculate diff for this page;
    set protection mode of this page to PROT_READ;
    set cache state of this page to RO;
    free twin;
  }
  send DIFF messages;
  wait until all DIFFGRANT messages are received;
  for (all RW local pages)
  {
    add this page to wtnt;
    set protection mode of this page to PROT_READ;
  }

```

그림 6.4: ProcessRWPages() 알고리즘

그림 6.4는 임계영역에 들어 갈 때와 벗어날 때, RW 상태의 페이지들을 처리하는 함수인 ProcessRWPages()의 코드를 보여준다. 먼저 RW 상태인 모든 캐쉬 페이지들에 대해서 다음과 같은 일을 해야한다: 해당 페이지 번호를 *wtnt*에 추가하고, diff를 계산해서 저장한다. 페이지의 보호 모드를 읽기전용으로 바꾸고, 캐쉬 상태를 RO로 바꾸고, 이전에 생성된 twin 영역을 없앤다. DIFF 메시지를 사용해서 저장된 diff를 각 페이지의 홈 프로세스에게 전달하고 DIFFGRANT 메시지가

오기를 기다린다. 다음으로, RW 상태인 모든 지역 페이지들에 대해서 해당 페이지 번호를 *wtnt*에 추가한 후 페이지의 보호 모드를 읽기전용으로 바꾼다.

Input: lockid

Output: none

Algorithm:

```
ProcessRWPages(wtnt);  
push wtnt into stack;  
send an ACQ message to home process of the lock;  
wait until an ACQGRANT(in_wtnt) message is received;  
invalidates all cache pages according to in_wtnt;
```

그림 6.5: DsmLock() 알고리즘

그림 6.5은 DsmLock() 위한 알고리즘을 보여준다. 먼저 ProcessRWPages() 함수를 호출하여 현재 RW 상태인 모든 페이지들을 처리하고, write notice(*wtnt*)를 스택에 저장한다. lock의 홈 프로세스에게 ACQ 메시지를 보낸다. ACQGRANT 메시지가 도착할 때까지 대기한 후, 메시지가 도착하면 함께 온 write notice 정보(*in_wtnt*)에 따라서 해당 캐쉬 페이지들을 무효화시킨다.

Input: lockid

Output: none

Algorithm:

```
ProcessRWPages(wtnt);  
send a REL(wtnt) message to home process of the lock;  
pop stack (saved_wtnt);  
append current write notices to save_wtnt;  
set save_wtnt as current write notices;
```

그림 6.6: DsmUnlock() 알고리즘

그림 6.6는 DsmUnlock() 위한 알고리즘을 보여준다. 먼저 현재 임계영역에서 수정된 모든 페이지들을 ProcessRWPages() 호출을 통해 처리한다. 그 결과로써 만들어지는 write notice 정보(*wtnt*)를 REL 메시지와 함께 lock의 홈 프로세스에게 보낸다. 스택에 저장된 이전 write notice 정보(*saved_wtnt*)를 복구한다. 내부 임계영역에서의 수정은 외부 임계영역에서의 수정이라고 볼 수 있기 때문에 내부 임계영역의 write notice를 외부 임계영역의 write notice에 추가시킨다.

그림 6.7은 lock의 홈에서 ACQ 메시지가 들어왔을 때 호출되는 AcqServer() 함수의 코드를

```

Input: ACQ message
      pid   : pid of requesting process;
      incar  : incarnation number of requesting process;
Output: none
Algorithm:
  cur_incar = current incarnation number of lock manager;
  if (request queue is empty)
  {
    insert (pid, incar) into request queue;
    append all write notices from incar to cur_incar to wtnt;
    send an ACQGRANT(wtnt) message to the requesting process;
  }
  else
    insert (pid, incar) into request queue;

```

그림 6.7: AcqServer() 알고리즘

보여준다. *pid*는 ACQ 메시지를 보낸 프로세스의 ID이고, *incar*은 그 프로세스가 현재 보고 있는 incarnation 번호를 의미한다. 만약 request queue가 비었다면, 즉 현재 lock을 가지고 있는 프로세스가 없다면 *pid* 프로세스에게 바로 ACQGRANT 메시지를 보낼 수 있다. 먼저 request queue에 (*pid*, *incar*) 쌍을 삽입한 후, *pid* 프로세스가 아직 보지 못한 incarnation의 모든 write notice를 ACQGRANT 메시지와 함께 보낸다. 만약 request queue가 비어있지 않다면, 현재 누군가가 lock을 가지고 있으므로 단순히 request queue에 (*pid*, *incar*) 쌍을 삽입하기만 한다.

```

Input: REL message
      in_wtnt : write notice included in REL message;
Output: none
Algorithm:
  cur_incar = current incarnation number of lock manager;
  increment cur_incar;
  save in_wtnt;
  delete the first (pid, incar) item from request queue;
  if (request queue is not empty)
  {
    pid   = pid of the first item in request queue;
    incar = incarnation number of the first item in request queue;
    append all write notices from incar to cur_incar to wtnt
    send an ACQGRANT(wtnt) message to process pid;
  }

```

그림 6.8: RelServer() 알고리즘

그림 6.8은 lock의 홈에 REL 메시지가 도착했을 때 호출되는 RelServer() 함수의 코드를 보여 준다. 앞서 얘기했듯이 lock의 incarnation은 lock이 release될 때마다 1씩 증가한다. 따라서 lock manager는 현재 incarnation 번호를 1 증가시키고, 증가된 incarnation 번호와 REL 메시지와 함께 온 write notice 정보(*in_wtnt*)를 저장한다. request queue의 첫번째 항목을 없앤다. 만약 request queue가 비었다면 lock을 얻고자 하는 프로세스가 없는 것이므로 그대로 종료한다. 만약 request queue가 비지 않았다면, queue의 첫번째 항목에 저장된 프로세스에게 ACQGRANT 메시지를 전송한다. 이때 그 프로세스가 보지 못한 incarnation에 대한 write notice(*wtnt*)를 함께 전송한다.

3 Barrier의 구현

ScC에서 barrier는 전체 프로그램을 포함하는 전역적인 consistency scope로 생각할 수 있다 (즉, 가장 바깥에 존재하는 임계영역으로 생각할 수 있다.) 따라서 barrier에서 떠나는 것은 consistency scope를 *open*하는 것으로 볼 수 있고, barrier에 도착하는 것은 consistency scope를 *close*하는 것으로 볼 수 있다.

Input: none

Output: none

Algorithm:

```

ProcessRWPages(wtnt);
send a BARR(wtnt) message to barrier server;
wait until a BARRGRANT(in_wtnt) message is received;
invalidates all cache pages according to in_wtnt;

```

그림 6.9: DsmBarrier() 알고리즘

실제 barrier의 구현은 그림 6.9와 같다. 먼저 전역 scope에서 수정된 모든 페이지들을 ProcessRWPages() 함수를 호출해서 처리한다. 그 결과로 얻어진 write notice(*wtnt*)를 BARR 메시지와 함께 미리 정해진 barrier 서버로 전송한 후, BARRGRANT 메시지가 오기를 기다린다. BARRGRANT 메시지가 도착하면 함께 온 write notice 정보(*in_wtnt*)에 따라서 해당 캐시 페이지들을 무효화시킨다.

```

Input: BARR message
       in_wtnt : write notices included in a BARR message;
Output: none
Algorithm:
  increment barrier_count;
  append in_wtnt to wtnt;
  if (barrier_count is equal to the number of processes)
    send a BARRGRANT(wtnt) message to all processes;

```

그림 6.10: BarrServer() 알고리즘

그림 6.10은 barrier 서버에서 BARR 메시지가 도착했을 때 호출되는 BarrServer() 함수의 알고리즘을 보여준다. barrier 서버는 BARR 메시지가 도착할 때마다 *barrier_count*를 1 증가시키고, BARR 메시지에 함께 온 write notice 정보를 *wtnt*에 누적시킨다. *barrier_count* 값이 현재 병렬 프로그램에 참가하고 있는 프로세스의 수와 일치하면 모든 프로세스가 barrier에 도착한 것이므로 각 프로세스에 BARRGRANT 메시지를 보낸다. BARRGRANT 메시지와 함께 누적된 write notice 정보(*wtnt*)도 함께 보낸다.

제 7 장

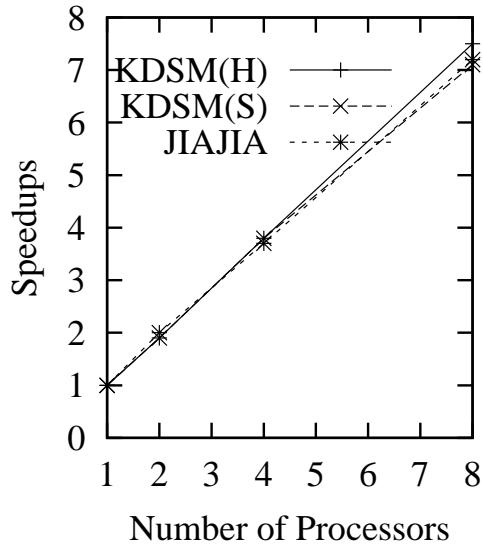
성능 측정

이 장에서는 각각 HLRC와 ScC 프로토콜을 사용했을 때의 KDSM 시스템의 실행 성능에 관해서 설명한다. 성능 측정은 500 MHz Pentium III 프로세서를 탑재하고 128M 의 메모리를 가진 8대의 PC를 100 Mbps Switched Fast Ethernet으로 묶은 클러스터 시스템에서 수행되었다. 성능 측정을 위한 응용 프로그램은 SPLASH2 [12] 의 Water와 LU, Rice 대학의 SOR, TSP 를 사용하였다. 다른 시스템과의 성능비교를 위해 Chinese Academy of Science에서 개발된 JIAJIA [6] 시스템을 비교 대상으로 삼았다. JIAJIA는 ScC [1] 모델을 사용하는 소프트웨어 DSM 시스템이다.

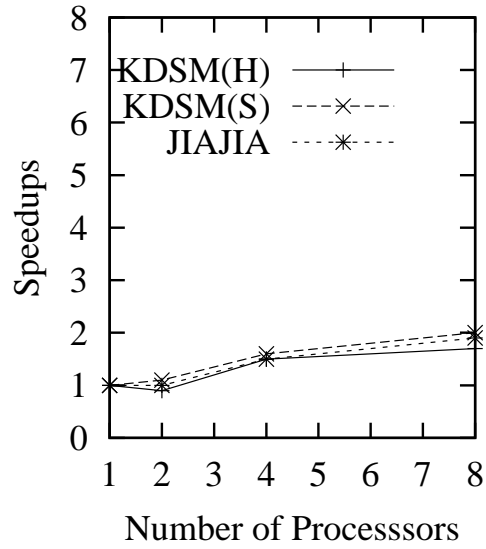
표 7.1은 각 응용에 대해서 프로세서의 수를 1 개, 2 개, 4 개, 8 개씩 사용했을 때 걸린 실행 시간을 보여준다. KDSM(H)로 표시한 항목은 HLRC로 프로토콜을 사용한 결과이고, KDSM(S)로 표시한 항목은 ScC 프로토콜을 사용한 측정 결과이다.

그림 7.1은 각 응용 프로그램에 대해서 얻을 수 있는 속도 향상을 보여준다.

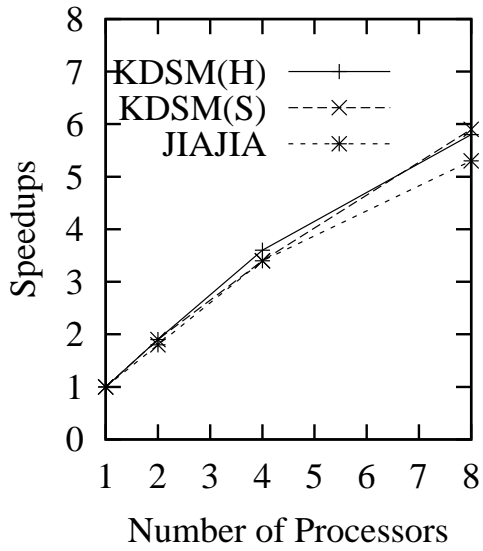
모든 프로그램에 대해 KDSM은 HLRC, ScC 버전 모두 유사한 성능을 보였다. 또한 JIAJIA와 비교하였을때 거의 유사하거나 약간 나은 성능을 보였다. SOR과 WATER의 경우 그다지 좋은 성능향상을 얻지 못하였는데 이는 메시지의 처리로 인해 barrier 와 lock 에서의 대기 시간이 길어지기 때문이다. 이것은 소프트웨어 DSM 에서 일반적으로 나타나는 문제로서 앞으로 해결해야할 과제이다.



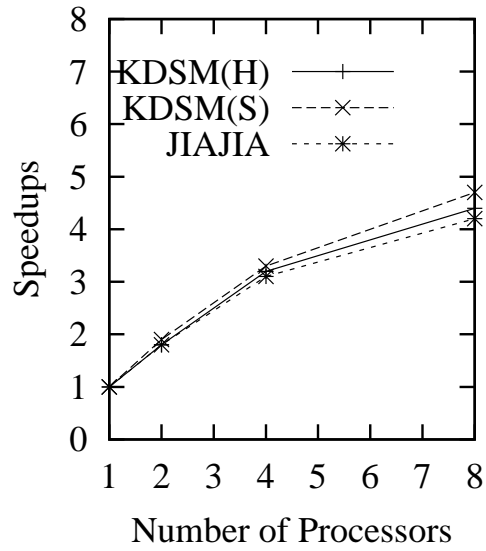
(a) LU 1024x1024



(b) SOR 1024x1024



(c) TSP 20 cities



(d) WATER 1728

그림 7.1: 속도 향상 결과

application	system	processor count			
		1	2	4	8
LU 1024×1024	KDSM(H)	161.49	86.00	42.34	21.79
	KDSM(S)	161.11	86.33	42.62	22.85
	JIAJIA	161.48	80.68	43.85	22.41
SOR 1024×1024	KDSM(H)	28.95	31.95	19.79	17.14
	KDSM(S)	28.81	26.49	18.09	14.60
	JIAJIA	30.17	29.10	20.46	16.22
TSP 20 cities	KDSM(H)	93.84	48.82	26.01	16.25
	KDSM(S)	93.20	48.82	27.09	15.71
	JIAJIA	93.18	50.58	27.43	17.51
WATER 1728 moles	KDSM(H)	104.04	56.52	32.09	23.76
	KDSM(S)	104.26	57.30	32.44	22.68
	JIAJIA	104.50	56.85	33.26	25.10

표 7.1: 실행 시간

제 8 장

결론

본 보고서에서는 프로토타입 분산 공유메모리 시스템인 KDSM에 대해서 설명하였다. KDSM 시스템의 특징은 페이지 기반 무효화 프로토콜과 홈 기반 프로토콜을 사용해서 HLRC 및 ScC 두 개의 memory consistency model을 지원한다는 것이다. KDSM 시스템은 Linux 운영체제 상에서 사용자 수준 프로세스로 구현되었고, 기반 통신 구조로 TCP/IP를 사용한다. 100 Mbps Fast Ethernet으로 연결된 Pentium III PC 상에서 성능을 측정해 본 결과, 메시지의 양이 적은 응용은 성능 향상이 잘 되는 반면, 메시지의 양이 많은 응용은 오히려 성능 저하가 일어나기도 했다. 기반 통신 구조를 JIAJIA 시스템의 것으로 교체해서 성능을 측정한 결과, 네 개의 응용중 세 개(LU, TSP, Water)는 성능이 JIAJIA에 필적하였고, 한 개는(SOR) 성능이 그리 좋지 않았다.

본 보고서에는 다루지 못한 내용 중, 전반적으로 TCP/IP의 성능이 나쁜 이유에 대한 분석과 SOR의 경우 KDSM 시스템이 JIAJIA에 비해 메시지 수와 양이 많은 이유에 대한 분석이 필요할 것으로 생각된다.

KDSM 시스템은 성능을 염두에 두지 않고 개발된 프로토타입 DSM 시스템이다. 그럼에도 불구하고 LU나 Water의 경우, 성능이 어느 정도 향상되었다는 점과 TCP가 아닌 UDP를 썼을 때 성능이 JIAJIA에 필적할 정도란 점은 아주 고무적이었다. 여기에, 여러 가지 소프트웨어 분산 공유메모리에 적용할 수 있는 성능 향상 기법, 예를 들어서 홈을 이동시키는 방법 등을 적용시켜 KDSM 시스템의 성능을 향상시킬 여지가 아주 많다고 생각한다. 또한 아주 큰 문제점으로 지적된

TCP/IP의 성능을 개선시키기 위해서 현재 Fast Ethernet이 아닌 Myrinet 네트워크 상에서 사용자 수준 통신을 할 수 있는 통신 계층을 개발 중에 있다. 사용자 수준 통신 계층 위에서 돌아가는 KDSM 버전이 나오면 그 성능은 지금보다 훨씬 좋을 것이라 예상된다.

저서 목록

- [1] L. Iftode, J.P. Singh, and K. Li. Scope Consistency: A Bridge between Release Consistency and Entry Consistency. In *Proceedings of the 8th Annual ACM Symposium on Parallel Algorithms and Architectures*, June 1996.
- [2] K. Gharachorloo, D. Lenoski, P. Gibbons, A. Gupta, and J. Hennessy. Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors. In *Proceedings of the 17th Annual International Symposium on Computer Architectures*, June 1990.
- [3] Y. Zhou, L. Iftode, and K. Li. Performance Evaluation of Two Home-Based Lazy Release Consistency Protocols for Shared Virtual Memory Systems. In *Proceedings of the second USENIX Symposium on Operating System Design and Implementation(OSID'96)*, October 1996.
- [4] B.N. Bershad, M.J. Zekauskas, and W.A. Sawdon. The Midway Distributed Shared Memory System. In *Proceedings of the IEEE COMPCON '93 Conference*, February 1993.
- [5] J.B. Carter, J.K. Bennett, and W. Zwaenepoel. Implementation and Performance of Munin. In *Proceedings of the 13th ACM Symposium on Operating System Principles*, February 1991.
- [6] W. Hu, W. Shi, and Z. Tang. The JIAJIA Software DSM System. Technical report, Institute of Computing Technology, Chinese Academy of Sciences, February 1998.

- [7] K. Li. *Shared Virtual Memory on Loosely Coupled Multiprocessors*. PhD thesis, Yale University, Department of Computer Science, September 1986.
- [8] C. Amza, S. Dwarkadas, P. Keleher, A. Cox, and Z. Zwaenepoel. Treadmarks: Shared Memory Computing on Networks of Workstations. *IEEE Computer*, 29(2), February 1996.
- [9] W. R. Stevens. *Advanced Programming in the UNIX Environment*. Addison Wesley, 1998.
- [10] P. Keleher. *Distributed Shared Memory Using Lazy Release Consistency*. PhD thesis, Rice University, December 1994.
- [11] J. Singh, W. Weber, and A. Gupta. SPLASH: Stanford Parallel Applications for Shared Memory. *Computer Architecture News*, 20(1), March 1992.
- [12] S. Woo, M. Ohara, E. Torrie, J. Singh, and A. Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proceedings of the 22th Annual Symposium on Computer Architecture*, 1995.