Journal of the Korea Academia-Industrial cooperation Society
Vol. 16, No. 5 pp. 3399-3406, 2015

http://dx.doi.org/10.5762/KAIS.2015.16.5.3399
ISSN 1975-4701 / eISSN 2288-4688

NRD 접근을 위한 리눅스 블록 디바이스 드라이버

손태영[1], 임성락[1*]
[1]호서대학교 컴퓨터공학부

Block Device Driver of Linux for Accessing the NRD

Tae-Yeong Son[1], Seong-Rak Rim[1*]
[1]**Division of Computer Engineering, Hoseo University**

요 약 NRD(Network RamDisk)는 원격 시스템의 메모리를 네트워크를 통하여 마치 자신의 블록 디바이스처럼 사용할 수 있도록 하는 기법이다. 기본적으로 이 기법은 NRD 접근을 요청하는 NRD 클라이언트와 NRD를 제공하는 NRD 서버 시스템으로 구성된다. 본 논문에서는 리눅스 커널(2.6) 수준에서 NRD 접근을 지원하기 위한 블록 디바이스 드라이버의 설계, 구현 그리고 실험을 제시한다. 이를 위하여 우선 기존 리눅스 커널에서 블록 디바이스에 대한 접근 요청이 처리되는 과정을 분석하여 NRD를 지원하기 위하여 요구되는 추가적인 기능들을 도출한다. 그리고 이 기능들을 제공하는 NRD 클라이언트의 디바이스 드라이버와 NRD 서버를 설계 및 구현한다. 마지막으로 NRD 서버 시스템을 구축하고, 구현된 NRD 디바이스 드라이버를 통한 NRD 클라이언트의 NRD 접근 요청을 실험함으로써 제시한 기법의 기능적 타당성을 검토한다.

Abstract NRD(Network RamDisk) is a scheme which allows a system to use the memory of the remote systems just as his own block device via networking. Basically, it consists of a client requesting an NRD access and server providing the NRD. In this paper, we describe the design, implementation and experiment of the block device driver for accessing the NRD in the Linux kernel(2.6) level. First of all, we have analyzed the flow of processing the requests for accessing the block devices in the traditional Linux kernel and figured out the additional functions required for supporting the NRD. Then we have designed and implemented the device diver of NRD client and NRD server for providing these functions. Finally, we have established a NRD server system, and reviewed its functional feasibility by experimenting the requests of NRD access through the NRD device driver implemented on a NRD client.

Key Words : Block Device Driver, NRD(Network RamDisk), Request Function

1. 서론

블록 디바이스는 블록(block) 단위로 접근하기 때문에 오늘날 대부분의 컴퓨터 시스템은 데이터 저장장치로 사용하고 있다. 그러나 하드 디스크와 같은 블록 디바이스의 경우, 탐색시간(seek time)이 CPU의 처리속도에 비해 너무 오래 걸리기 때문에 보다 효율적인 블록 디바이스 접근 기법에 대한 연구가 계속되어 오고 있다 [1,2,3].

NRD(Network RamDisk)[4]는 원격 시스템의 메모리를 네트워크를 통해 마치 자신의 블록 디바이스처럼 접근할 수 있도록 하는 기법으로서 블록 디바이스를 필요로 하는 임베디드 시스템에서 커널의 수정 없이 커널 수준의 NRD 디바이스 드라이버 접근이 가능하여 매우 유용하게 사용할 수 있다. 이 기법은 기본적으로 NRD 서버와 NRD 클라이언트 시스템으로 구성된다. NRD 서버는 자신의 메모리 주소 공간의 일부분을 NRD로 제공한다. 한편 NRD 클라이언트는 사용자 프로세스의 NRD

*Corresponding Author : Seong-Rak Rim(Hoseo Univ.)
Tel: +82-41-540-5708 email: srrim@hoseo.edu
Received February 4, 2015 Revised (1st March 20, 2015, 2nd April 1, 2015)
Accepted May 7, 2015 Published May 31, 2015

I/O 요청을 서버로 전달하고, 서버의 처리 결과를 사용자 프로세스로 전달한다.

본 논문에서는 리눅스 환경에서 NRD를 지원하기 위한 블록 디바이스 드라이버를 제시하고자 한다. 이를 위하여 리눅스 커널(2.6)에서 블록 디바이스에 대한 I/O 요청이 처리되는 과정을 커널 수준으로 분석하여 NRD 접근을 지원하기 위해 요구되는 추가적인 기능을 도출한다.

마지막으로 제시한 NRD 블록 디바이스 드라이버의 기능적 타당성을 검토하기 위하여 NRD 서버 시스템과 NRD 클라이언트 시스템을 실험용 임베디드 보드[5]에 구축하여 NRD 접근을 실험 한다.

2. 관련 연구

리눅스 커널에서 사용자 프로세스의 블록 디바이스에 대한 I/O 요청이 처리되는 과정은 Fig. 1과 같다[6,7].

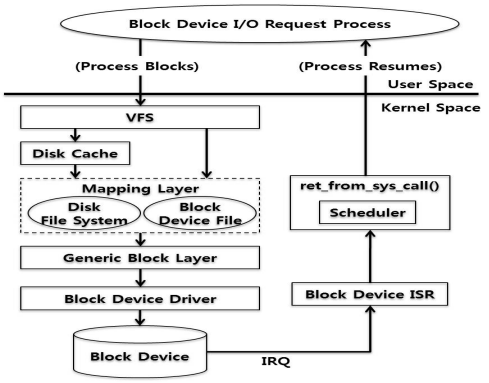


Fig. 1. Flow of Block Device I/O Request

사용자 프로세스로부터 블록 디바이스에 존재하는 어떤 파일에 대한 읽기 요청이 발생할 경우, 커널 내부의 VFS(Virtual File System)에서 요청된 블록의 데이터가 디스크 캐시(Disk Cache)에 존재 하는지 검사한다. 만약 존재하면 요청한 프로세스에게 곧바로 전달하지만 존재하지 않을 경우, 블록 디바이스 드라이버에게 위임하고 대기 상태(TASK_INTERRUPTIBLE)에서 기다리게 된다.

블록 디바이스 드라이버는 블록 디바이스 제어기에게 '읽기' 명령을 보냄으로써 블록 디바이스로부터의 읽기 작업이 시작된다. 블록 디바이스 제어기는 읽기 작업을 완료한 후, 해당 블록 디바이스의 IRQ(Interrupt Request)를 발생시켜 작업 완료를 커널에게 통지한다.

블록 디바이스의 IRQ가 발생되면 해당 블록 디바이스의 ISR(Interrupt Service Routine)은 읽기 완료 통지와 관련된 일련의 작업을 수행한 후, ret_from_sys_call()을 호출한다. ret_from_sys_call()에서는 블록 디바이스로부터 읽기 작업을 기다리고 있던 대기 상태의 프로세스를 준비 상태(TASK_RUNNING)로 변환시키고 스케줄러를 호출함으로써 프로세스의 실행이 재개된다.

이러한 일련의 처리과정에서 리눅스 커널은 블록 디바이스를 효율적으로 관리하기 위하여 모든 블록 디바이스마다 자신의 요청 큐(Request Queue)를 유지한다. 요청 큐는 해당 블록 디바이스에 대한 프로세스의 I/O 요청을 보관하기 위한 자료구조로, 사용자 프로세스로부터 I/O 요청을 커널이 해당 블록 디바이스의 요청 큐에 등록한다. 요청 큐에 등록된 I/O 요청은 커널 블록 디바이스 스레드(kblockd)가 실행될 때 요청 큐에 연결된 디바이스 드라이버의 요청 처리함수(request_fn())에 의해 처리된다. 요청 처리함수는 디바이스 드라이버가 커널에 등록될 때 요청 큐에 연결된다.

NRD 접근을 지원하기 위해서는 Fig. 2와 같이 사용자의 입출력 요청을 받는 블록 디바이스 드라이버를 포함하고 있는 클라이언트와 자신의 메모리 공간을 블록 디바이스로 사용할 수 있도록 서비스하는 서버가 필요하다[8].

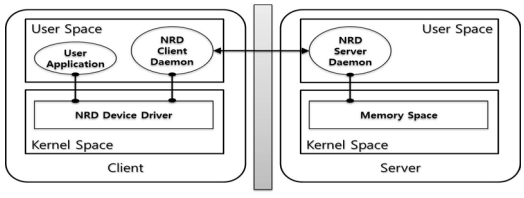


Fig. 2. Structure of NRD

클라이언트의 NRD 디바이스 드라이버는 사용자 응용 프로그램의 I/O 요청을 받아 사용자 영역에 있는 NRD 클라이언트 데몬 프로그램으로 전달한다. NRD 클라이언트 데몬 프로그램은 전달받은 I/O 요청을 네트워크를 통해 NRD 서버 데몬 프로그램으로 전달한다. 서버의 NRD 서버 데몬 프로그램은 메모리 공간을 블록 디바이스처럼 이용할 수 있도록 클라이언트에게 서비스를 제공한다.

리눅스 커널에서 사용자 프로세스의 블록 디바이스에 대한 I/O 요청이 처리되는 과정을 분석한 결과, NRD 접근을 지원하기 위해서는 NRD 디바이스 드라이버의 요청 처리함수가 요구됨을 도출 하였다. 한편 NRD 접근을 지원하기 위한 요청처리 함수는 처리완료를 통지하기 위한 NRD 디바이스의 IRQ 및 ISR, 그리고 NRD 서버와의 통신을 위한 네트워크 기능이 추가적으로 요구된다. 따라서 이러한 기능들을 제공하는 NRD 클라이언트의 디바이스 드라이버와 NRD 서버를 설계한다.

3. 설계

본 논문에서 제시한 NRD 클라이언트와 NRD 서버의 기본적인 동작흐름은 Fig. 3과 같다.

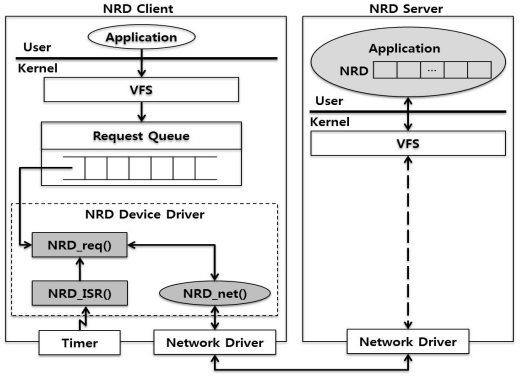


Fig. 3. NRD Server and NRD Client

NRD 서버 프로세스는 자신의 메모리 일부분을 NRD로 선언하고 NRD 클라이언트의 NRD 접근 요청을 기다린다. NRD 클라이언트의 요청 처리함수(NRD_req())는 요청 큐에 등록된 요청을 네트워크 스레드(NRD_net())를 통하여 NRD 서버에게 전달하고, NRD 서버로 부터 처리된 결과를 수신한 후 타이머 IRQ에 의해 NRD_ISR()에서 처리 완료를 커널에게 통지한다. 통지를 마친 NRD_ISR()는 요청 큐의 다음 요청을 처리하도록 NRD_req()를 호출한다.

3.1 NRD 클라이언트

3.1.1 요청 처리함수/처리완료 통지

요청 큐에 등록된 요청들은 순차적으로 하나씩 처리되어야 한다. 즉 하나의 요청이 처리되고 있는 동안에 요청 처리함수는 처리완료 통지될 때까지 기다려야한다. 이를 위하여 Fig. 4와 같이 요청 처리함수(NRD_req())와 처리완료를 통지(NRD_ISR()) 간에 공유변수(flag)를 사용하여 동기화한다.

NRD_req()는 공유변수(flag)가 설정되어 있지 않을 때 요청 큐로부터 요청을 가져오고 요청 명령(R/W)에 따라 NRD 서버에게 전송할 메시지를 생성한다. 생성한 메시지가 네트워크 스레드(NRD_net())에 의해 NRD 서버로 전송되도록 대기 큐(NRD_net_wq)에 등록된 NRD_net()를 깨운다.

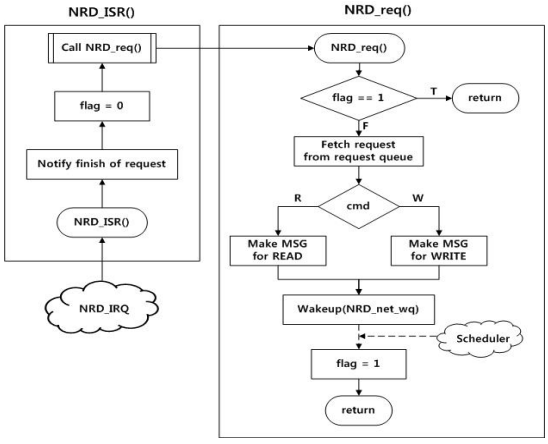


Fig. 4. NRD_req() & NRD_ISR()

NRD_net()가 깨어나면 NRD_req()은 NRD_net()의 우선순위에 의해 선점된다. 스케줄러에 의해 재개된 NRD_req()는 flag를 설정함으로써 NRD_ISR()에 의해 처리완료 통지 후 flag가 해제될 때까지 기다린다.

NRD_ISR()는 NRD_net()에서 읽기/쓰기 요청에 대한 응답에 경우 활성화된 타이머 IRQ 발생으로 실행된다. NRD_ISR()는 요청 큐에서 가져온 하나의 요청이 처리완료 되었음을 알리고 flag를 해제함으로써 NRD_req()로 하여금 다음 요청을 처리하도록 한다.

3.1.2 네트워크 스레드

기존의 NRD 접근에는 서버에게 요청 메시지를 전송하고 그 결과 값을 수신하기 위해 사용자 영역의 네트워크 데몬 프로그램을 이용한다. 이와 같은 경우 커널 영역에서 사용자 영역으로 잦은 데이터 복사는 임베디드 시스템의 속도 저하를 가져온다.

이러한 문제를 해결하기 위해서 커널 영역에서 네트

워크 통신을 할 수 있는 ksocket[9]을 이용하여 네트워크 스레드 함수(NRD_net())를 Fig.5와 같이 설계한다.

NRD_net()는 NRD_req()에 의한 선점을 방지하기 위하여 자신의 우선순위를 상향 설정한다. 커널 수준의 네트워크를 위한 ksocket을 생성한 후, 대기 큐(NRD_net_wq)에 등록되어 NRD_req()에 의해 깨어날 때까지 기다린다.

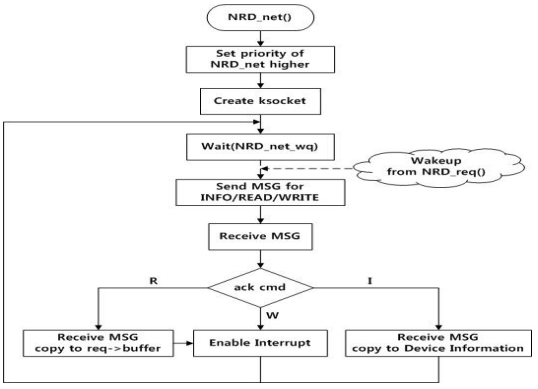


Fig. 5. NRD_net()

NRD_req()에 의해 재개되면 NRD 서버로 요청 메시지를 전송하고 이에 대한 응답 메시지를 수신한다. 정보(I)에 대한 응답 메시지일 경우에는 수신된 데이터를 디바이스 정보로 복사하고 다시 대기 큐에 등록되어 기다린다. 읽기(R)에 대한 응답 메시지일 경우 수신된 데이터를 요청 버퍼로 복사하고, 요청 처리완료 통지를 위한 타이머 IRQ를 활성화한 후 다시 대기 큐에 등록되어 기다린다. 쓰기(W)에 대한 응답 메시지일 경우에는 요청 처리완료 통지를 위한 타이머 IRQ를 활성화한 후 다시 대기 큐에 등록되어 기다린다. 이와 같이 NRD_req(), NRD_ISR() 그리고 NRD_net()가 상호 협력하여 NRD 접근 요청을 처리한다.

3.2 NRD 서버

NRD 서버는 다음과 같은 NRD 클라이언트의 요청 메시지에 대한 서비스를 제공하기 위하여 Fig. 6과 같이 설계한다.

① 정보(I) : NRD 디바이스 드라이버 등록에 필요한 NRD에 대한 물리적 정보.
② 읽기(R) : NRD에 대한 물리적인 읽기.
③ 쓰기(W): NRD에 대한 물리적인 쓰기.

NRD 서버는 NRD로 사용될 메모리 공간을 선언하고 NRD 클라이언트로부터 서비스 요청을 받기 위해 소켓을 생성한 후, NRD 클라이언트로부터 요청 메시지를 기다린다.

정보 요청(I)인 경우에는 NRD의 정보를 응답 메시지로 생성한다. 읽기 요청(R)인 경우에는 NRD에서 데이터를 읽어 응답 메시지를 생성한다.

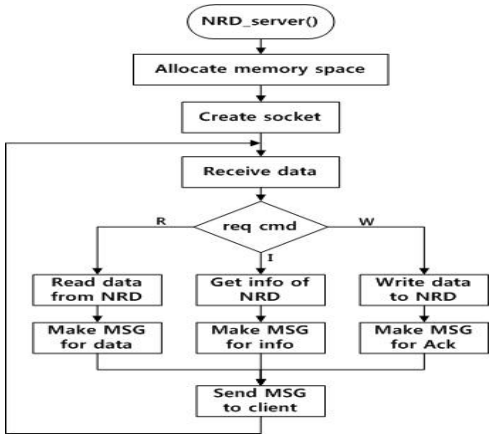


Fig. 6. NRD_server()

쓰기 요청(W)인 경우에는 요청 메시지의 데이터를 NRD에 쓰고 응답 메시지를 생성한다. 마지막으로 생성된 모든 응답 메시지를 NRD 클라이언트에게 전송하고 다음 요청 메시지를 기다린다.

4. 구현

4.1 NRD 클라이언트

NRD 클라이언트에 추가적으로 요구되는 기능(NRD_req(), NRD_IRQ&ISR(), NRD_net())을 두 개의 모듈 프로그램으로 분리하여 구현한다.

4.1.1 NRD 디바이스 드라이버 모듈

NRD 디바이스 드라이버 모듈의 init() 와 NRD_ISR()는 Fig. 7과 같이 구현한다.

NRD 디바이스 드라이버 모듈의 init()에서는 NRD 디바이스 드라이버를 사용하기 위한 등록과정을 수행한다. register_blkdev()로 NRD 디바이스 드라이버를 등록

하고, NRD 디바이스의 정보를 등록하기 위해서 정보 요청(I)메시지를 생성한다. wakeup()로 NRD_net()를 재개시키고 NRD 서버로부터 NRD 디바이스 정보를 응답을 받은 후 alloc_disk()로 gendisk 구조체를 할당받아 NRD 디바이스의 정보를 설정한다. NRD 디바이스의 요청 큐는 blk_init_queue()로 요청 큐를 생성하고 NRD_req()를 연결한다. NRD 디바이스 정보를 입력한 gendisk 구조체는 add_disk()로 NRD 디바이스를 커널에 등록한다.

```
init( ) {
  register_blkdev(MAJOR, NAME);
  ⋮ Make MSG for INFO
  wakeup(NRD_net_wq);
  alloc_disk(NRD_MAX_PARTITIONS);
  blk_init_queue(NRD_req, &lock);
  add_disk(gendisk);
  init_timer(&(NRD_timer));
  NRD_timer.function = NRD_ISR;
  return;
}

NRD_ISR( ) {
  __blk_end_request_cur(req, 0);
  flag = 0;
  NRD_req(q);
}
```

Fig. 7. Module init() & NRD_ISR()

요청 처리 완료 통지를 위해 init_timer()로 타이머를 초기화하고, NRD_ISR()를 등록한다.

NRD_ISR()는 __blk_end_request_cur()로 커널에게 요청이 처리 되었다는 것을 통지한다. NRD_req()와 동기화를 위해 flag 변수를 0으로 설정하고, NRD_req()를 호출하여 다음 요청을 처리하도록 한다.

NRD 디바이스 드라이버 모듈의 NRD_req()는 Fig. 8과 같이 구현한다.

```
NRD_req(struct request_queue * q) {
  if(flag)   return;
  req = blk_fetch_request(q);
  switch(req->cmd) {
    case READ  : // Make MSG for READ
      break;
    case WRITE : // Make MSG for WRITE
      break;
  }
  wakeup(NRD_net_wq);
  flag = 1;
  return;
}
```

Fig. 8. NRD req()

NRD_req()는 flag 변수가 1로 설정되어 있다면 동작을 종료하고, 0으로 설정되어 있다면 blk_fetch_request()로 요청 큐에서 요청의 정보를 가져온다. 요청 큐에서 가져온 요청의 명령(READ/WRITE)에 따라 요청 메시지를 생성한다.

요청 메시지를 NRD 서버에게 전송하기 위하여 대기 큐(NRD_net_wq)에 등록된 NRD_net()을 깨운다. NRD_net()가 깨어나면 NRD_req()은 NRD_net()의 우선순위에 의해 선점된다. 스케줄러에 의해 재개된 NRD_req()는 flag를 설정함으로써 NRD_ISR()에 의해 처리완료 통지 후 flag가 해제될 때까지 기다린다.

4.1.2 커널 네트워크 모듈

커널 네트워크 모듈의 init() 및 NRD_net()를 Fig. 9와 같이 구현한다.

```
init( ) {
  kthread_run(NRD_net, NULL, "NRD_net");
}

NRD_net(void *data) {
set_user_nice(current, -20);
ksocket(client_ksocket);
  while (1) {
    wait_event_interruptible(NRD_net_wq);
    ksendto(buff_tx);
    krecvfrom(buff_rx);
    switch(buff_rx->cmd) {
      case I  : // Recv MSG copy Device Info
        break;
      case R : // Recv MSG copy req->buffer
        memcpy(req->buffer, buff_rx->data);
        NRD_timer.expires = jiffies + HZ/10;
        add_timer(&NRD_timer);
        break;
      case W : // Timer enable
        NRD_timer.expires = jiffies + HZ/10;
        add_timer(&NRD_timer);
        break;
    }
  }
}
```

Fig. 9. Module init() & NRD_net()

커널 네트워크 모듈의 init()에서는 kthread_run() 으로 커널 스레드(NRD_net())를 생성한다.

NRD_net()는 NRD_req()에 의한 선점을 방지하기 위하여 set_user_nice()으로 자신의 우선순위를 상향 설정한다. 커널 수준의 소켓을 ksocket()으로 생성한 후, wait_event_interruptible()로 대기 큐(NRD_net_wq)에 등록되어 NRD_req()에 의해 깨어날 때까지 기다린다.

NRD_req()에 의해 재개되면 ksendto()로 NRD 서버

에게 요청 메시지를 전송하고 이에 대한 응답 메시지를 krecvfrom()로 수신한다.

정보(I)에 대한 응답 메시지일 경우에는 수신된 데이터를 디바이스 정보로 복사하고 다시 대기 큐에 등록되어 기다린다. 읽기(R)에 대한 응답 메시지일 경우 수신된 데이터를 memcpy()로 요청 버퍼로 복사하고, 요청 처리완료 통지를 위한 타이머 IRQ를 add_timer()로 활성화한 후 다시 대기 큐에 등록되어 기다린다. 쓰기(W)에 대한 응답 메시지일 경우에는 요청 처리완료 통지를 위한 타이머 IRQ를 활성화한 후 다시 대기 큐에 등록되어 기다린다.

4.2 NRD 서버 응용 프로그램

NRD 접근을 지원하기 위한 NRD 서버는 Fig. 10과 같이 구현한다.

```
#define NRD_INFO
char NRD[SIZE];
main( ) {
  socket();
  while(1) {
   recvfrom(buff);
   switch(buff->cmd) {
    case I : // Make MSG of ack for INFO
     break;
    case R : // Make MSG of ack for read
     memcpy(buff->data, NRD);
     break;
    case W : // Make MSG of ack. for write
     memcpy(NRD, buff->data);
     break;
    }
   sendto(MSG);
  }
}
```

Fig. 10. Process for NRD server

NRD 서버는 NRD의 물리적인 정보(NRD_INFO)와 NRD로 사용될 메모리 공간(NRD[SIZE])을 선언하고 NRD 클라이언트로부터 서비스 요청을 받기 위해 socket()로 소켓을 생성한 후, NRD 클라이언트로부터 요청 메시지를 recvfrom()로 수신한다.

정보 요청(I)인 경우에는 NRD_INFO를 참조하여 응답 메시지로 생성한다. 읽기 요청(R)인 경우에는 memcpy()로 NRD에서 데이터를 읽어 응답 메시지를 생성한다. 쓰기 요청(W)인 경우에는 요청 메시지의 데이터를 memcpy()로 NRD에 쓰고 응답 메시지를 생성한다. 마지막으로 각각의 생성된 모든 응답 메시지를 NRD 클라이언트에게 sendto()로 전송하고 다음 요청 메시지를 기다린다.

5. 실험

제시한 기법의 타당성을 검토하기 위하여 NRD 서버의 응용 프로그램과 NRD 클라이언트의 커널 네트워크 모듈 및 NRD 디바이스 드라이버 모듈을 Fig. 11과 같은 리눅스 임베디드 보드에 각각 탑재하여 다음과 같은 단계로 실험한다.

[단계-1] NRD 클라이언트 모듈 삽입
[단계-2] NRD 디바이스 마운트
[단계-3] NRD 디바이스 접근

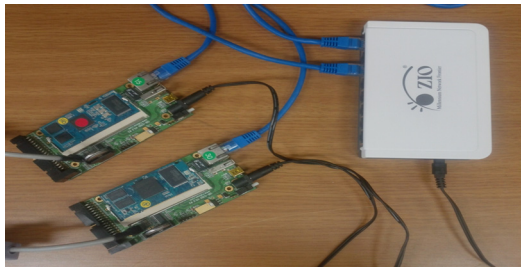


Fig. 11. Experimental Environment

5.1 NRD 클라이언트 모듈 삽입 [단계-1]

NRD 클라이언트의 커널 네트워크 모듈(kernel_network.ko)을 Fig. 12와 같이 삽입하고(②), ps 명령을 통해서 init()에 의해서 생성된 NRD_net를 확인한다(③). 이때, 커널 네트워크 모듈은 ksocket 라이브러리 함수를 사용하기 때문에 ksocket 모듈(ksocket.ko)을 먼저 삽입한다(①).

```
[root@falinux fafs]$ insmod ksocket.ko ①
ksocket version 0.0.2
BSD-style socket APIs for kernel 2.6 developers
msn : song.xian-guang@hotmail.com
blog: http://sxg.cublog.cn
[root@falinux fafs]$ insmod kernel_network.ko ②
FAFS MASTER MODULE V0.1
----------------------
[root@falinux fafs]$ ps
  PID USER       VSZ STAT COMMAND
    1 root      1600 S    init [3]
    2 root         0 SW   [kthreadd]
    3 root         0 SW   [ksoftirqd/0]
    4 root         0 SW   [watchdog/0]
    5 root         0 SW   [events/0]
    6 root         0 SW   [khelper]
 1279 root         0 SW   [flush-1:0]
 1345 root         0 SW<  [NRD_net]
 1350 root      2924 R    ps ③
[root@falinux fafs]$
```

Fig. 12. Init of kernel network module

NRD 디바이스 접근을 위한 NRD 디바이스 드라이버 모듈을 Fig. 13과 같이 insmod 명령으로 삽입하고(①), cat /proc/partitions 명령을 통해서 커널에 등록된 NRD 디바이스를 확인한다(②).

```
[root@falinux fafs]$ insmod NRD_device_driver.ko ①
NRD DEVICE DRIVER MODULE V0.1
================

 NRDa:
 unknown partition table
[root@falinux fafs]$ cat /proc/partitions
major minor  #blocks  name

  31        0       4096 mtdblock0
  31        1      14336 mtdblock1
  31        2     505856 mtdblock2
 240        0       4096 NRDa ②
[root@falinux fafs]$
```

Fig. 13. Init of NRD device driver module

5.2 NRD 디바이스 마운트 [단계-2]

NRD 클라이언트에서 NRD 디바이스를 사용하기 위해서 우선 Fig. 14와 같이 mknod 명령으로 NRD 디바이스 파일(NRDa)을 생성하고(①), ls 명령으로 블록 디바이스 파일을 확인한다(②).

```
[root@falinux fafs]$ mknod /dev/NRDa b 240 0 ①
[root@falinux fafs]$ ls /dev/NRD*
/dev/NRDa ②
```

Fig. 14. Creation of NRD device file

생성한 NRD 디바이스 파일을 통해 Fig. 15와 같이 mkfs 유틸리티를 사용하여 NRD 디바이스를 포맷한다(①). mount 명령으로 NRD 디바이스를 디렉토리(/mnt/hdd)에 마운트 하고(②), mount 명령으로 확인한다(③).

```
[root@falinux fafs]$ mkfs.vfat /dev/NRDa ①
[root@falinux fafs]$ mount /dev/NRDa /mnt/hdd/ ②
[root@falinux fafs]$ mount
rootfs on / type rootfs (rw)
/dev/root on / type ext2 (rw,relatime,errors=co
/proc on /proc type proc (rw,relatime)
/sys on /sys type sysfs (rw,relatime)
/dev/NRDa on /mnt/hdd type vfat (rw,relatime,fl
xed,errors=remount-ro) ③
[root@falinux fafs]$
```

Fig. 15. Format and Mount of NRD device

5.3 NRD 디바이스 접근 [단계-3]

NRD 디바이스를 마운트한 디렉토리에 파일 생성과 파일 읽기를 통해 NRD 디바이스에 쓰기와 읽기 요청을 발생시켜 구현한 NRD_req()와 NRD_ISR 및 NRD_net()의 동작을 확인한다.

5.3.1 NRD 디바이스 쓰기

NRD 디바이스가 마운트 된 디렉토리에 Fig. 16과 같이 cat 명령을 통해 문자열을 가지는 파일(test_file)을 생성하고(①), ls 명령으로 확인한다(②).

```
[root@falinux hdd]$ ls
[root@falinux hdd]$ cat >> test_file ①
nrd read write test

[root@falinux hdd]$ ls
test_file ②
[root@falinux hdd]$
```

Fig. 16. Write of NRD device

NRD 디바이스가 마운트되어 있는 디렉토리에 파일을 생성하면 NRD 디바이스로 쓰기 요청이 발생하고 NRD 디바이스에 파일 데이터가 저장된다.

5.3.2 NRD 디바이스 읽기

5.3.1에서 생성한 파일을 바로 읽으면 디스크 캐시에 있는 데이터가 전달되기 때문에 Fig. 17과 같이 umount 명령을 통해 NRD 디바이스의 마운트를 해제하고(①), mount 명령으로 NRD 디바이스를 디렉토리(/mnt/hdd)에 다시 마운트 한다(②).

```
[root@falinux mnt]$ umount /mnt/hdd/ ①
[root@falinux mnt]$ ls /mnt/hdd/
[root@falinux mnt]$ mount /dev/NRDa1 /mnt/hdd/ ②
[root@falinux mnt]$ ls /mnt/hdd/
test_file
[root@falinux mnt]$
```

Fig. 17. Unmount NRD device

5.3.1에서 생성했던 파일을 Fig. 18과 같이 cat 명령으로 읽으면 NRD 디바이스로 읽기 요청이 발생하여 파일의 데이터가 화면에 출력된다.

```
[root@falinux hdd]$ cat test_file

nrd read write test
[root@falinux hdd]$
```

Fig. 18. Read of NRD device

6. 결론

본 논문에서는 리눅스 환경에서 인터넷에 연결된 다른 시스템의 메모리 공간의 일부분을 마치 자신의 HDD

처럼 접근할 수 있는 기법을 제시하였다. 제시한 기법은 기본적으로 자신의 메모리 공간을 NRD로 제공하는 NRD 서버와 NRD를 마치 자신의 HDD처럼 접근하는 NRD 클라이언트로 구성된다.

이를 위하여 리눅스 커널(2.6)에서 HDD와 같은 블록 디바이스에 대한 사용자 프로세스의 접근 요청이 처리되는 과정을 분석하여 NRD 접근을 지원하기 위해 필요한 요구사항을 도출한 후, NRD 서버와 클라이언트의 NRD 디바이스 드라이버 모듈을 설계 구현하였다.

제시한 기법은 기존의 리눅스 커널 수정 없이 커널 수준의 NRD 디바이스 드라이버에 접근이 가능하도록 한다.

References

[1] Matthews, J. N., Roselli, D., Costello, A. M., Wang, R. Y., & Anderson, T. E., "Improving the perfomance of long-structured file system with adaptive methods", In Proc 16-th Symposium on Operating Systems Principles, Vol.31, No.5, pp.238-251, 1997.

[2] Rosenblum, M., Ousterhout, J. K., "The design and implementation of a log-structured file system", ACM Transactions on Computer Systems (TOCS), Vol.10, No.1 pp.26-52, 1992.
DOI: http://dx.doi.org/10.1145/146941.146943

[3] Seltzer, M., Bostic, K., McKusick, M. K., & Staelin, C., "An implementation of a long-structured file system for unix", Proceedings of the USENIX Winter 1993 Conference Proceedings on USENIX Winter 1993 Conference Proceedings, USENIX Association, pp.3-3, 1993.

[4] Flouris, Michail D., and Evangelos P. Markatos., "The network RamDisk: Using remote memory on heterogeneous NOWs", Cluster computing, Volume 2, No.4, pp.281-293, 1999.
DOI: http://dx.doi.org/10.1023/A:1019051330479

[5] Falinux(주), "Processor Module", http://www.falinux.com/kr/desktop/product/epb/sib_g100, 2015

[6] Bovet, Daniel P., and Marco Cesati., Understanding the Linux Kernel, O'Reilly Media, 2005.

[7] Nutt, Gary J. Kernel projects for linux. Addison Wesley Longman, 2000.

[8] Jun-Ho Her, "Design and Implementation of Reliable Network RamDisk", Korea Information Processing Society(KIPS), Vol.8, No.2, pp.283-286, 2001.

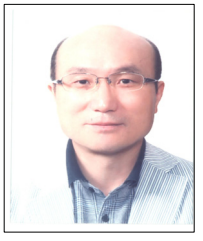
[9] Beta-song's project - ksocket, "ksocket", ksocket.sourceforge.net, 2015

손 태 영(Tea-Yeong Son) **[정회원]**

- 2004년 2월 : 호서대학교 컴퓨터공학과 (학사)
- 2013년 2월 : 호서대학교 일반대학원 컴퓨터공학과 (석사)
- 2013년 3월 ~ 현재 : 호서대학교 컴퓨터공학과 박사과정

<관심분야>
임베디드 시스템, 분산 파일시스템, 리눅스 커널

임 성 락(Seong-Rak Rim) **[정회원]**



- 1983년 2월 : 서울대학교 공과대학원 컴퓨터공학과 (공학석사)
- 1992년 8월 : 서울대학교 공과대학원 컴퓨터공학과 (공학박사)
- 1983년 3월 ~ 1990년 2월 : ㈜금성반도체연구소 선임연구원
- 1993년 3월 ~ 현재 : 호서대학교 컴퓨터공학과 교수

<관심분야>
임베디드 시스템, 리눅스 커널