모바일 플래시 파일 시스템 - MJFFS

김 영 관* · 박 현 주*

A Mobile Flash File System - MJFFS

Young-Gwan Kim* · Hyun-Ju Park*

Abstract

As the development of an information technique, gradually, mobile device is going to be miniaturized and operates at high speed. By such the requirements, the devices using a flash memory as a storage media are increasing. The flash memory consumes low power, is a small size, and has a fast access time like the main memory. But the flash memory must erase for recording and the erase cycle is limited. JFFS is a representative filesystem which reflects the characteristics of the flash memory. JFFS to be consisted of LSF structure, writes new data to the flash memory in sequential, which is not related to a file size. Mounting a filesystem or an error recovery is achieved through the sequential approach. Therefore, the mounting delay time is happened according to the file system size.

This paper proposes a MJFFS to use a multi-checkpoint information to manage a mass flash file system efficiently. A MJFFS, which improves JFFS, divides a flash memory into the block for suitable to the block device, and stores file information of a checkpoint structure at fixed interval. Therefore mounting and error recovery processing reduce efficiently a number of filesystem access by collecting a smaller checkpoint information than capacity of actual files. A MJFFS will be suitable to a mobile device owing to accomplish fast mounting and error recovery using advantage of log foundation filesystem and overcoming defect of JFFS.

Keywords : Flash, File System, Mobile, Checkpoint, LSF

논문접수일 : 2003년 8월 25일 논문게재확정일 : 2004년 2월 25일
* 한밭대학교 정보통신컴퓨터공학부

1. 서 론

반도체 기술의 발전은 다양한 모바일 기기(mobile device)의 발전을 가져왔다. 그러나 모바일 기기는 크기가 제한되어 있고, 외부의 충격, 격심한 온도변화 등 불안한 환경에서 동작하여야 하기 때문에 시스템 설계시 많은 어려움이 따른다. 최근 모바일 장비의 저장 매체로써 플래시 메모리(flash memory)의 사용이 증대되고 있다. 이런 플래시 메모리 사용의 증대는 디스크에 비해 다음과 같은 장점을 가지기 때문이다[2]. 첫째, 플래시 메모리는 기존의 자기 디스크와 같은 회전식 자기 매체가 아니라 메모리이기 때문에 빠른 접근 속도를 가지므로 높은 성능을 제공한다. 둘째, 플래시 메모리는 외부 충격에 강하고 비휘발성(non-volatile)이다. 셋째, 저 전력 구동이 가능하고 크기가 작다. 플래시 메모리의 일반적인 특징은 <표 1>과 같다[7].

〈표 1〉 플래시 메모리 특성

Read Cycle	70 ~ 110ns
Write Cycle	12 μs/byte (Typical)
Erase Cycle	0.5 ~ 1s /block
Erase Cycles limit	100,000 cycles
Erase Unit Size	8 ~ 64 kbytes
Power Consumption	8~55 mA(active state) 7~250 μA (standby state)

플래시 메모리는 두 가지 단점이 있는데, 그 중 하나는 다시 쓰기 기능이 없다는 것이다. 따라서 데이터를 기록한 영역을 다시 사용하기 위해서는 반드시 삭제를 해야 한다. 플래시 메모리의 삭제는 EU(erase unit) 단위로 이루어지는데 0.5~1초 정도의 시간이 걸린다. 또 다른 단점은 플래시 메모리를 지우는 횟수가 100,000번 정도로 제한되어 있는 점이다[4]. 이러한 플래시 메모리의 단점을 극복하고 효과적으로 사용하기 위해 다양한 삭제 정책(cleaning policy)과 wearleveling 기법이 적용된 파일시스템이 제안되고 있다.

현재 연구된 플래시 메모리 파일시스템으로는 FTL(flash translation layer), TrusFFS, JFFS (The Journalling Flash File System), QplusFFS 등이 있다. FTL은 순차적인 플래시 공간이 디스크의 섹터(sector)처럼 보이도록 하기 위해 매핑(mapping) 관리를 수행하는 드라이버 형식으로 구현되어 있다. TrusFFS는 VxWorks RTOS에 맞게 구현한 시스템으로 플래시에 대한 블록 디바이스 인터페이스를 제공하기 위해 Block-to-Flash 전환 시스템을 사용한다. JFFS는 플래시 공간을 순차적으로 보고, 발생되는 데이터를 크기에 관계없이 순차적으로 저장하는 LSF (log-structured file system) 방법의 하나이다. QPlusFFS는 QPlus에서 사용되는 파일시스템으로 한국과학기술원과 한국전자통신연구원이 공동 개발하였다. QPlusFFS는 기존의 파일시스템을 지원하도록 FAT(file allocation table) 형식으로 파일을 저장하며, 이를 위해 주소 매핑을 위한 FML(flash memory mapping layer) 계층을 추가로 적용하였다[1].

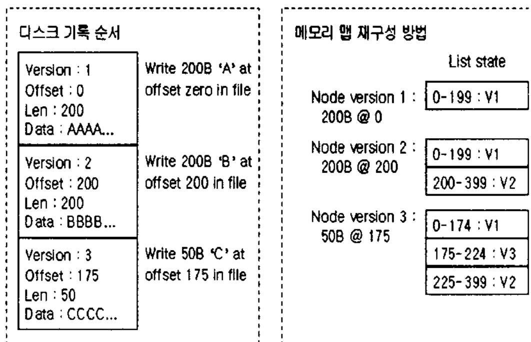
이 중 플래시 메모리에 대표적으로 사용되는 것이 JFFS이다. JFFS는 LFS 방법을 적용하여 wearleveling과 오류회복에 강한 파일시스템이다. 그러나 JFFS 파일시스템의 마운트와 오류회복 과정을 수행하는 시간이 파일시스템에 순차적인 접근을 통해 이루어지므로 파일시스템의 사용량에 관계없이 전체 용량에 따라 비례하여 증가되는 단점이 있다. 본 논문은 JFFS 파일시스템을 개선한 MJFFS(multi-checkpoint JFFS)에 대한 특징과 구현에 대해 기술한다. MJFFS는 다중 체크포인트 정보를 이용하여 마운트와 오류회복에 따른 지연 시간을 감소시켰다. 논문의 구성은 다

음과 같다. 2장에서는 JFFS(The Journalling Flash File System)에 대해 알아보고, 3장에서 제안하는 멀티-체크포인트 플래시 파일시스템을 기술하고, 4장에서 멀티-체크포인트 플래시 파일시스템의 구현 예와 성능에 대해 알아보고, 마지막으로 5장에서 결론과 추후 과제를 분석한다.

2. JFFS

JFFS는 1999년 Axis Communications에서 공개 하였으며, 저널링(Journalling)이란 로그에 기반한 데이터 저장 방식을 사용하는 파일시스템이다.

저널링 기법은 <그림 1>의 (a)와 같은 단일 순차적 쓰기 기법을 통해 쓰기 작업의 성능을 향상시켰다. 즉, 수정시 기존의 파일을 직접 수정하는 것이 아니라 수정된 내용을 로그 형태로 저장한다. 그리고 그 파일을 메모리로 읽어 들일 때, <그림 1>의 (b)와 같이 저장된 로그의 버전 정보를 이용하여 파일을 수정한다[6].

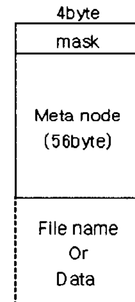


(a) 순차적인 데이터 기록 (b) 메모리에서 재배치된 로그정보

〈그림 1〉 Journaling (Log-structured) filesystem

데이터를 저장할 때 로그 데이터를 순차적으로 기록하므로 따로 Wearleveling에 대한 구현을 할 필요가 없다. 또한 갑작스럽게 전원이 꺼져도, 플래시 메모리의 공간을 순차적으로 읽음으로써 파일시스템을 복구할 수 있다[3, 5].

JFFS가 플래시 메모리에 기록하는 파일 노드는 <그림 2>처럼 메타 데이터와 파일이름 또는 데이터를 포함하는 구조를 가진다. 메타 데이터는 일반적인 아이노드번호, 부모의 아이노드, 버전(version), 생성/변경시간들, 오프셋(offset), 데이터 크기(dsize), 체크섬(checksum) 등을 포함한다. JFFS의 데이터 부분의 크기는 메타 데이터에 나타난 파일 이름의 길이와 데이터의 크기에 따라 변하게 된다. 그러므로 JFFS의 파일 노드는 각각 파일의 속성 값에 따라 그 길이가 다르다.



〈그림 2〉 JFFS data node

JFFS는 마운트 될 때, 파일시스템을 처음부터 마지막까지 순차적으로 로그 정보를 살펴보면서 파일시스템의 구조를 파악한다. 이 순차적인 접근 과정을 통해 메모리 맵을 구성하여 파일시스템의 전체적인 정보를 유지한다. 마운트 과정이 끝난 후, 추가(로그)로 기록되는 파일 정보를 메모리 맵에 적용시킴으로써 항상 최신 정보를 유지한다.

JFFS 파일시스템을 사용하는 도중 시스템의 전원이 갑자기 꺼져도 다시 순차적인 접근을 통해 파일시스템을 재구성하게 된다. 그러므로

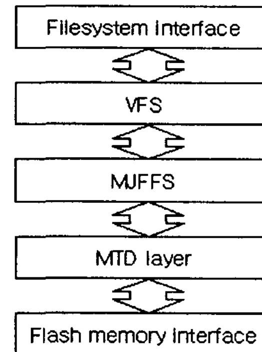
JFFS의 오류회복 과정은 파일시스템을 다시 마운트 하는 과정으로 이루어진다.

JFFS는 파일시스템을 구성하기 위해서 파일시스템 전체를 순차적으로 다시 읽어야 하므로 Mount 시간이 길어지는 단점이 있다. 마운트 지연시간은 파일시스템의 전체적인 크기에 비례하여 증가되는데 이는 적은 공간을 사용하는 경우에도 전체 파일시스템을 모두 읽어야 마운트가 가능하다. 또한 오류가 발생하여 오류회복에 걸리는 시간도 마운트 시간과 유사하다. 파일시스템을 마운트하기 위한 순차적인 접근으로 생성되는 파일정보를 저장할 메모리 공간도 파일시스템 전체에 대해 유지해야 하기 때문에 파일시스템의 크기에 비례하여 증가된다. 본 논문에서 제안하는 MJFFS는 JFFS의 이러한 단점을 보완하여 마운트 시간과 오류회복 시간을 단축시켰으며, 메모리의 사용을 동적으로 수행하여 필요한 경우 메모리를 반환하는 방식이다.

3. MJFFS(Multi-checkpoint JFFS)

MJFFS(multi-checkpoint JFFS)는 JFFS v1.0 파일시스템의 단점을 수정 보완한 것이다. JFFS의 물리적인 저장 방법을 변경하여 고정 블록으로 하며, 멀티체크포인트를 사용하여 고정된 단위만큼의 파일시스템 파일과 디렉터리에 대한 정보를 반복 저장한다.

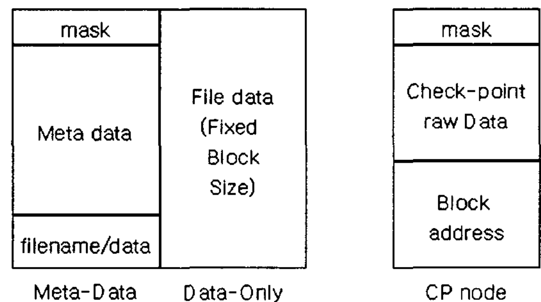
본 논문에서 구현한 MJFFS 파일시스템의 구조를 <그림 3>에 나타내었다. 상위 계층은 VFS를 통해 파일시스템 인터페이스 계층에 연결된다. UNIX의 인터페이스인 VFS를 지원하므로 기존의 응용 프로그램을 그대로 구동시킬 수 있는 확장성을 가지고 있다. 또한 UNIX나 리눅스에 익숙한 사용자에게 편의성과 일관성을 제공한다. 하단의 MTD(memory technology device) layer는 리눅스에서 일반적인 플래시 메모리나 RAM과 같은 다양한 메모리 장치를 쉽고, 편리한 지원을 가능하게 하며, 일반적으로 커널에 포함되어 있다.



<그림 3> 전체적인 구조

3.1 플래시 메모리 데이터

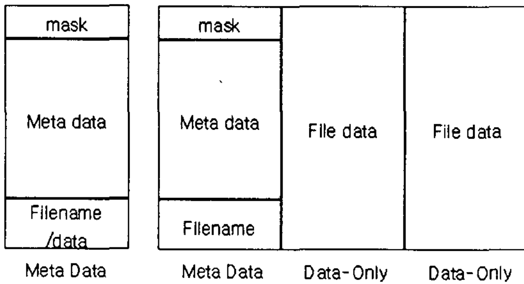
MJFFS는 플래시메모리에 물리적으로 기록하는 단위를 고정된 크기의 블록으로 나누어 저장한다. 플래시 메모리에 저장되는 데이터 타입을 <그림 4>에 나타내었다. 데이터 타입은 (a) 데이터 노드(data node)와 (b) 체크포인트 노드(checkpoint : CP)로 나뉜다. 데이터 노드에는 메타데이터가 포함된 메타-데이터 블록(meta-data block)과 오직 데이터만을 기록하는 데이터-온리 블록(data-only block)이 있다.



<그림 4> MJFFS의 2가지 데이터 타입

메타데이터 필드는 파일에 대한 메타 정보를 기록하기 위한 데이터 필드이다. JFFS에서 사용하는 메타데이터와 유사한 메타데이터를 사용하는데, 이 메타데이터 정보를 이용하여 메타-데이터 노드에 데이터가 포함된 노드와 포함되지 않은 노드를 구분한다.

파일의 정보와 파일 데이터를 메타-데이터 블록과 데이터-온리 블록에 저장한다. 파일 저장 방법은 2가지이다. 첫 번째는 <그림 5>의 <a>와 같이 하나의 블록으로 데이터 노드가 구성된다. 이것은 파일 이름과 저장하려는 데이터의 크기가 작아 하나의 블록에 저장된다. 이 경우 메타데이터 필드에 블록에 데이터가 포함되었음을 알리는 정보가 기록된다. 두 번째는 <b>와 같이 저장될 데이터의 크기가 큰 경우 메타-데이터 블록은 메타데이터 필드와 파일 이름만을 저장하고 다음 블록에 파일의 데이터를 기록한다. 이것이 데이터-온리 블록이며, 데이터-온리 블록은 연속적으로 나타날 수 있지만, 항상 가장 처음 블록은 메타-데이터 블록이다.



(a) Data size가 작을때 (b) Data size가 클때

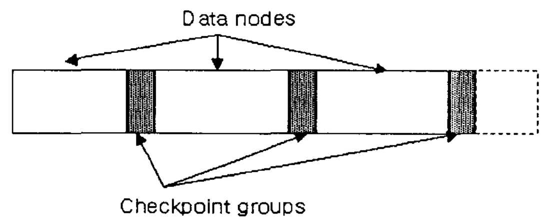
〈그림 5〉 데이터 노드의 구성

다음은 메타-데이터 블록과 데이터-온리 블록을 구별하여 저장할 때 사용할 함수이다. 파일이름의 길이(nsize)와 데이터의 길이(dsize)의 합이 남아있는 공간(avail_space)보다 크면 0을 반환하고, 작으면 1을 반환한다.

```
is_include()
{
    if(nsize > 0)
        return ((nsize+dsize) > avail_space) ? 0 : 1;
    return (dsize > avail_space) ? 0 : 1;
}
```

3.2 Multi-checkpoint 구조

체크포인트 그룹(checkpoint group)은 반복적으로 지정된 위치에 있는 체크포인트 그룹부터 다음 위치의 체크포인트 그룹 사이에 변경된 파일의 정보를 저장한다. 체크포인트 그룹은 파일에 대한 변경사항을 기록한 파일 체크포인트와 디렉터리의 변경사항을 기록한 디렉터리 체크포인트의 집합으로 구성된다. <그림 6>은 체크포인트 그룹과 데이터 노드의 관계를 간단히 나타낸 것이다.



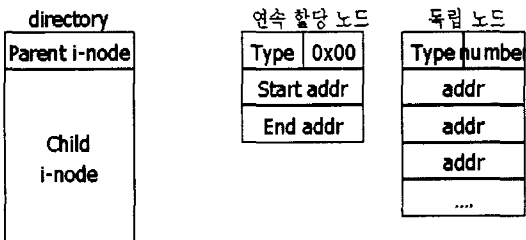
〈그림 6〉 Checkpoint nodes의 위치

체크포인트 그룹은 항상 고정된 위치에 기록된다. 데이터 노드를 기록할 때, 기록될 데이터가 체크포인트 위치를 넘어가게 되면, 체크포인트 위치까지 데이터를 나누어 기록하고, 체크포인트 그룹을 기록한 후 남아있는 데이터를 다시 기록한다.

체크포인트 노드는 <그림 4>의 (b)와 같이 체크포인트 데이터 필드와 블록 주소 필드로 구

성된다. 체크포인트 데이터 필드는 체크포인트가 가리키는 파일의 위치, 자신과 부모의 아이노드 번호, 체크포인트 정보의 갱신여부를 나타내는 정보를 가지고 있다. 파일의 위치 정보는 파일이 처음 생성되어 저장된 위치와 파일이름이 갱신된 위치, 그리고 마지막으로 저장된 위치에 대한 주소를 저장한다. 블록 주소 필드는 디렉터리와 파일을 구분하여 다르게 사용하는데 디렉터리 체크포인트는 부모와 자식들의 주소를, 파일은 파일이 저장된 블록의 주소를 저장한다. 따라서 체크포인트 정보를 획득하면 파일시스템의 전체적인 파일 메모리 맵을 구성할 수 있다.

다음은 체크포인트 노드의 데이터 부분이다. 첫 번째로 디렉터리에 대한 체크포인트일 경우에 <그림 7>의 (a)와 같이 부모 디렉터리의 아이노드 값을 저장할 필드와 자식의 아이노드 값을 저장할 필드로 구성된다. 두 번째로 파일에 대한 체크포인트인 경우에는 <그림 7>의 (b)와 같이 연속적으로 할당된 노드를 위한 구조와 독립적으로 할당된 노드를 가리키는 구조로 되어 있으며, 파일 체크포인트는 두 가지 구조를 적절히 혼합하여 할당된 노드를 표현한다. <그림 7>의 (b)에서 Type은 연속할당 노드와 독립 노드를 구분하기 위한 필드이고, 독립노드의 number는 기록된 노드의 수를 나타낸다.



(a) 디렉터리 CP (b) 파일 CP

<그림 7> Checkpoint Node의 데이터 필드 구조

다음 프로그램은 파일 노드의 데이터의 크기에 따라 세 가지의 경우로 처리된다. 첫 번째는 하나의 노드에 메타데이터와 파일 데이터를 기록한다. 두 번째와 세 번째의 경우는 메타데이터와 파일데이터를 나누어 기록하는 방식으로 그 데이터 노드가 체크포인트가 기록될 위치일 경우 나누어 기록한다. BLOCK_GROUP은 체크포인트가 기록될 간격을 저장하고, BLOCK_SIZE는 MJFFS가 사용하는 블록의 크기를 나타낸다. REGION_VALUE는 기록된 파일 노드와 체크포인트와의 남아있는 용량을 나타낸다. Write_node()를 호출한 후에는 체크포인트를 기록해야 하는지 점검한다.

```
Write_node (node, data, dsize)
{
   if(is_include()){
      REGION_VALUE = REGION_VALUE -
                          BLOCK_SIZE;
      return (jffs_file_node_write(DATA, node,
                          filedata, BLOCK_SIZE));
   }
   if (dsize + 512 > REGION_VALUE) {
      jffs_file_node_write(DATA, node, 0,
                          BLOCK_SIZE);
      jffs_file_node_write(DATA, node, filedata,
                 REGION_VALUE-BLOCK_SIZE);
      dsize = dsize - (REGION_VALUE +
                        BLOCK_SIZE);
      REGION_VALUE = BLOCK_GROUP;

      do_checkpoint_data_write();
      REGION_VALUE = REGION_VALUE -
                        (checkpoint-data_size)
      return Write_node (node, data, dsize);
   }
   REGION_VALUE = REGION_VALUE - dsize;
   return (jffs_file_node_write(DATA, node,
                          filedata, dsize));
}
```

3.3 MJFFS의 마운트

JFFS의 마운트 과정은 플래시메모리를 탐색해 나가면서 데이터가 기록된 부분을 찾는다. 데이터가 기록된 위치에서 플래시가 지워진 위치까지 파일데이터를 모두 읽고, 검사하고, 메모리 맵을 구성한다. 이후에 에러에 대한 복구를 수행한다. 모든 파일에 대한 작업이므로 파일 시스템의 크기와 파일시스템의 사용량에 따라 마운트의 시간이 달라진다.

MJFFS의 마운트 과정은 체크포인트 노드 정보를 획득함으로써 대부분이 이루어진다. 체크포인트는 자신의 아이노드, 부모의 아이노드, 자식의 아이노드, 파일의 위치와 같은 파일에 대한 기본 정보를 저장하고 있으므로 체크포인트 정보를 모두 종합하면 파일시스템의 전체적인 구조를 알 수 있다. MJFFS의 마운트는 다음의 알고리즘을 따른다.

```
mount_mjffs()
{
  // 체크포인트 그룹이 저장되어 있는 위치로 이동
  move_checkpoint_groug_position();
  // 파일시스템의 모든 체크포인트 그룹을 검색
  loop (is_checkpoint_group()){
  // 체크포인트 그룹에서
  // 마지막 체크포인트 노드까지 검색
    do {
      // 체크포인트 노드의 위치로 이동
      move_next_checkpoint_position();
      // 체크포인트 노드의 정보를 종합
      add_checkpoint_node_info();
      // 정보를 종합한 체크포인트 노드가
      // 마지막이면 loop 끝.
    } loop(!is_last_checkpointnode());
    // 다음 체크포인트 그룹의 위치로 이동.
    move_checkpoint_group_position();
  }
  // 만일 마지막 체크포인트 그룹에 에러가 있다면
  // 직전의 체크포인트 그룹 다음에 기록된 파일의
  // 위치로 이동
  if(last_checkpoint_group_error)
     move_file_node_previous_checkpoint_group();
  // 그렇지 않으면 체크포인트 그룹 이후에
  // 스캔 해야 할 파일의 위치로 이동
  else
     move_file_node_position();
  // 체크포인트 그룹 이후에 기록된 파일노드를
  // 모두 종합
  loop(is_file_node){
     if (is_error_free_file_node())
        add_file_node_info();
     move_next_file_node_position();
  }
}
```

이 과정이 끝난 후 메모리맵의 내용은 체크포인트로 기록되지 않은 파일에 대해서는 자세한 정보를 유지하고 있으며, 체크포인트로 기록된 파일들에 대해서는 기본 정보만을 유지하고 있다. 모든 정보를 유지한 파일에 대한 접근요청은 즉시 응답이 가능하지만, 기본 정보를 유지한 파일에 대해 접근 요청을 받으면 메모리맵을 바탕으로 해당 파일의 위치에 접근하여 메타데이터를 가져온 후 요청에 응답한다. 파일의 메타데이터 요청에 의한 접근 방법은 플래시메모리의 READ 연산이 일반 디스크 접근 속도만큼이나 빠르고, 1~3개의 블록에 대한 READ 연산을 수행함으로써 정보 획득이 가능하기 때문에 성능이 저하되지 않는다.

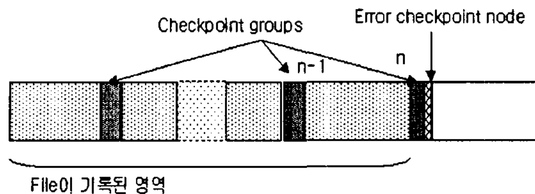
MJFFS가 초기 마운트 후에 파일시스템을 사용하는 과정에서 접근한 파일의 수에 따라 점차 메모리맵의 크기가 커진다. 시스템의 메모리 부족에 대비해 MJFFS는 필요한 경우 현재 사용중이거나 아직 체크포인트를 기록하지 않은 파일을 제외하고 나머지 파일에 대한 정보를 기본 정보만 남긴 후 메모리맵에서 삭제한다. 필요한 경우 초기 마운트 상태로 되돌아가는 이 기능은 시스템의 메모리 사용량을 증대시킨다.

3.4 오류복구 방법

디스크 기반의 저장장치에 비해 플래시메모리는 충격이나 자기 등의 영향이 적다. 플래시메모리의 사용에 있어서 오류는 갑작스럽게 전원이 차단되어 안정된 종료가 이루어지지 않았을 때 데이터 손실에 의한 오류이다. MJFFS는 오류의 발생 원인에 따라 몇 가지 오류 복구 방법을 제시한다.

3.4.1 체크포인트 오류

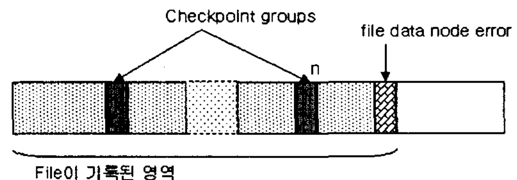
체크포인트 노드에 발생하는 오류는 두 가지 형태가 있다. 첫 번째는 <그림 8>과 같이 전체 파일시스템에서 마지막 체크포인트 노드를 기록하는 도중 발생된 오류이고, 두 번째는 전체 파일시스템의 체크포인트 그룹 중에서 중간 위치의 체크포인트 노드에 발생된 오류이다.



〈그림 8〉 파일시스템 마지막 체크포인트 오류

<그림 8>는 파일시스템의 마지막에 위치한 체크포인트 노드를 기록하는 과정에서 시스템의 이상으로 n번째 체크포인트 그룹의 마지막 체크포인트 노드가 기록이 완료되지 않은 상태이다. 이 파일시스템을 다시 마운트 하게 되면 정상적으로 기록되지 않은 체크포인트 노드의 불완전한 정보를 통합하게 된다. 이러한 경우 오류 복구는 MJFFS를 마운트 하는 과정에서 버전 n-1의 체크포인트 그룹 정보와 버전 n의 체크포인트 그룹 정보를 종합하는 과정에서 처리된다. 마운트 과정 중에 마지막 체크포인트 노드가 아니면 체크포인트 노드의 정보를 종합하며, 버전 n인 체크포인트 그룹이 마지막 체크포인트 그룹이라고 판단되고 체크포인트 그룹의 마지막 체크포인트 노드가 이상이 없으면 버전 n인 체크포인트 그룹의 정보를 추가한다. 그리고 버전 n인 체크포인트 그룹 이후에 기록된 데이터의 정보를 분석한다. 그러나 마지막 버전 n의 체크포인트 그룹을 분석하는 과정에서 에러를 발견하거나, 체크포인트 그룹의 마지막 노드임을 나타내는 비트가 TRUE로 된 노드가 없을 경우 체크포인트 그룹을 기록하는 과정에서 에러가 발생한 것으로 가정한다. 이러한 경우 버전 n-1인 체크포인트 그룹에서 버전이 n인 체크포인트 그룹 사이에 기록된 파일 정보를 스캔하여 종합한다. 이 정보를 이용하여 에러가 발생한 다음 노드의 위치에 체크포인트 노드를 추가로 기록하여 체크포인트 그룹을 완료시킨다. 후에 수행되는 마운트 과정에서는 오류가 있는 체크포인트 노드는 무시하며 다음 체크포인트 노드를 획득하여 종합한다.

두 번째는 전체 체크포인트 그룹의 중간 위치에서 에러가 있는 체크포인트 노드가 발견된 경우이다. 이 경우는 오류가 발생한 체크포인트 노드를 갱신하는 정보가 들어있는 새로운 체크포인트 노드가 다음 노드의 위치에 기록되어 있으므로 오류가 발생한 체크포인트 노드를 무시하고 다음의 체크포인트 노드를 획득한다.



〈그림 9〉 마지막에 위치한 파일의 오류

3.4.2 데이터 영역 기록시 오류

데이터 영역에서 오류는 <그림 9>와 같이 파일을 기록하는 도중에 시스템 이상으로 완전한

기록이 되지 않는 경우이다.

파일시스템을 마운트 하는 과정에서 체크포인트 그룹의 정보를 모두 종합한 후에 체크포인트 그룹 이후에 파일이 있는지 검사한다. 파일이 있는 경우 체크포인트 그룹으로 구성된 정보에 추가로 파일 정보 구성을 하는데, 마지막 체크포인트 그룹 이후의 모든 파일을 검사하면서 파일 정보를 추가한다. 오류가 있는 파일 블록을 만나게 되면 해당 블록을 무시하고 다음 블록을 읽는다. 오류 블록 이후에 파일데이터가 존재하면 블록 정보를 종합하고 없으면 마운트 과정을 완료한다. MJFFS는 로그로 추가되거나 변경되는 데이터를 기록한다. 따라서 하나의 데이터 블록에 오류가 발생하면 해당 블록 이전까지의 변경되거나 추가된 데이터는 유효하며, 오류가 발생한 해당 블록의 데이터만 무효화 된다.

3.4.3 MJFFS의 파일시스템 전체 오류 검사 기능

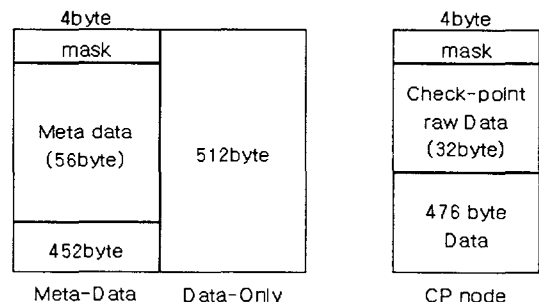
MJFFS는 필요한 경우에 파일시스템의 전체적인 오류 검사를 수행한다. 전체 오류 검사는 사용자의 요청이나 일정한 횟수의 마운트가 이루어지면 수행한다. fsck.mjffs로 명명되는 오류 검사 프로그램은 파일시스템의 처음부터 마지막까지 모든 파일의 checksum을 검사하며 파일시스템 정보를 새롭게 갱신한다. checksum이 일치되지 않는 파일은 새로운 로그 파일로 해당 블록이 빈값을 갖도록 새롭게 기록한다. 일반 자기 디스크보다 플래시메모리가 더 견고하기 때문에 파일시스템 전체 오류 검사는 빈번히 수행될 필요가 없다.

4. MJFFS의 구현 및 성능

4.1 MJFFS의 구현

JFFS와 MJFFS 파일시스템을 적용하기 위해서는 플래시 메모리를 사용하는 시스템이 필요하다. 그러나 두 파일시스템의 성능을 효율적으로 비교, 분석하기 위해 메모리를 플래시 메모리처럼 사용하는 에뮬레이터를 리눅스를 운영체제로 사용하는 PC에서 구현 하였다. 플래시 메모리를 대체하기 위해 사용자 영역의 힙 메모리를 4~8Mbyte를 할당하여 플래시 메모리처럼 사용하며, 실제 플래시 메모리의 READ, WRITE, ERASE 명령을 수행할 때 소요되는 시간과 동일한 성능으로 동작하도록 인터페이스를 구현하였다. 테스트 프로그램은 JFFS와 MJFFS를 별도로 제어하여 메모리에 동일한 데이터를 기록한 후에 각각 마운트에 소요되는 시간을 측정하였다.

MJFFS를 구현하기 위해 사용한 고정 블록의 단위는 512byte이다. 512byte는 JFFS에서 사용하는 메타데이터와 최대 파일 이름 길이, 그리고 로그 형태로 저장될 데이터 공간을 고려하여 정했다. <그림 10>은 블록의 크기가 512byte일 때, 데이터 노드와 체크포인트 노드의 구조를 나타낸 것이다.



〈그림 10〉 512byte 블록의 데이터 구조

데이터 노드의 메타-데이터 필드는 JFFS의 소스에서 사용하는 것과 동일하며, 메타-데이터 블록과 데이터-온리 블록을 구별하기 위한 1비트의 추가 사용만이 다르다. 메타-데이터 블록

에서 메타-데이터 필드와 마스크 필드를 제외한 452byte는 파일 이름이나 파일의 실제 데이터를 기록하는데 사용된다. JFFS와 MJFFS에서 지원하는 최대 파일이름의 길이는 255byte이다. 파일 이름과 실제 기록될 데이터의 합이 452byte보다 크면 파일 이름만 메타-데이터 블록에 저장하고, 나머지 데이터는 데이터-온리 블록에 저장한다.

<그림 10>의 (b) 체크포인트 노드의 체크포인트 로우 데이터 필드는 <표 2>와 같은 구조로 되어 있다.

〈표 2〉 Checkpoint Node 구조

Size	Field	Description
32 bit	first_node	해당 파일의 처음 위치
32 bit	name_node	해당 파일의 이름의 위치
32 bit	last_node	해당 파일의 마지막 노드
32 bit	inode	해당 파일의 inode 번호
32 bit	pion	부모의 inode 번호
16 bit	dsize	데이터의 크기
12 bit	spare	사용안함
1 bit	cp_group_end	그룹의 끝을 표현
1 bit	directory	디렉토리 체크포인트 표현
1 bit	append	추가 체크포인트 표현
1 bit	overwrite	체크포인트 정보 갱신
32 bit	checksum	오류 확인

<표 2>에서 first_node는 현재 체크포인트가 가리키는 파일의 시작 주소를 나타낸다. name_node는 파일의 이름이 있는 노드를 가르키며, rename에 의해 파일 이름이 변경되었다면 변경된 이름을 갖고 있는 노드의 주소 값이다. last_node는 마지막으로 수정된 노드의 주소를 가짐으로써 마지막으로 변경한 시간 정보를 빠르게 알 수 있다. inode와 pion는 자신의 i-node 값과 부모의 i-node 값을 저장하고 있다. cp_roup_end는 현재 기록되는 체크포인트가 체크포인트 그룹에서 마지막 체크포인트인가를 나타낸다. 체크포인트 노드가 디렉터리에 대한 체크포인트이면 directory 비트를 1로 하고, 파일에 대한 체크포인트이면 directory 비트를 0으로 한다. append 비트는 현재 체크포인트가 이전에 저장된 체크포인트 정보에 추가적인 정보일 경우 1의 값을 가지고, 그렇지 않으면 0이다. append 비트가 1일 때 first_node의 값은 이전에 기록된 체크포인트의 주소를 지정한다. overwrite 비트는 이전에 기록된 체크포인트의 정보를 무시하고 새롭게 기록되는 체크포인트임을 나타낸다.

체크포인트 그룹은 고정된 위치에 체크포인트 노드의 집합으로 구성된다. 체크포인트는 파일시스템에서 부가적으로 저장하는 정보이다. 체크포인트 정보가 없어도 파일시스템의 운영은 가능하지만, 전체 파일시스템에서 작은 용량을 할당하여 부가적인 정보를 저장함으로써 파일시스템의 접근 및 수행 속도를 향상시키는 것이 체크포인트 정보이다. 체크포인트 그룹을 사용하는 파일시스템의 최대 마운트 시간을 예측하기 위해, 16Mbyte의 플래시 메모리를 8Mbyte정도 사용한 파일시스템을 예로 체크포인트 그룹 이후에 읽어야 할 파일 데이터를 최대로 하여 계산하였다. 최대 마운트 시간의 계산은 <식 1>을 이용하였고, 계산의 편리성을 위해 체크포인트 그룹의 크기는 평균값을 이용하였다. 식의 결과는 파일시스템을 마운트하기 위해 읽어야할 총 데이터의 양과 플래시 메모리에서 데이터를 읽는 시간(T_{flash_read})의 곱으로 나타난다.

<식 1> $T_{sum} = T_{flash_read} \times (checkpoint\ group\ size) \times (checkpoint\ 개수) + T_{flash_read} \times (checkpoint\ group\ 이후에\ 읽을\ 데이터의\ 양)$

MJFFS 파일시스템에서 하나의 파일이 10Kbyte

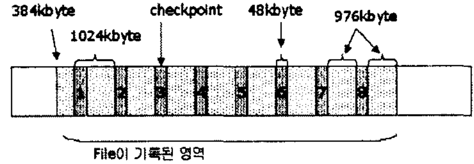
정도의 크기일 때 20개의 블록으로 표현이 가능하다. 그러므로 하나의 파일에 대해 하나의 체크포인트 노드만을 갖고 있으면 되기 때문에 1,024Kbyte 간격의 체크포인트 그룹의 크기는 대략 48Kbyte이며, 체크포인트 그룹 사이에 들어갈 파일의 수는 96개(파일 하나의 크기 : 10.16Kbyte)이다. <그림 11>과 같이 첫 번째 체크포인트 그룹 이전에 384Kbyte의 데이터가 있고, 8개의 체크포인트 그룹이 있으며, 마지막의 체크포인트 그룹 이후에 976byte의 파일 데이터가 있다. 마지막 데이터 기록 후에 아직 체크포인트 그룹을 저장하지 않은 상태가 최대 마운트 시간이다. 위 식을 이용하여 계산하면 초기 접근 시간은 다음과 같다.

$$T_{sum} = T_{flash_read} \times 48K \times 8 + T_{flash_read} \times 976k$$
$$= 1360k * T_{flash_read}$$

또한 최소 접근시간은 <그림 11>의 마지막에 체크포인트 그룹이 저장이 되었을 때이다. 이 경우는 체크포인트 그룹 9개에 대한 접근만으로 마운트가 가능하다.

$$T_{sum} = T_{flash_read} \times 48K \times 8 = 384k * T_{flash_read}$$

이 결과는 MJFFS 파일시스템에 접근할 때 최대 1360kbyte를 플래시 메모리에서 읽거나, 최소 384kbyte만 읽으면 마운트가 가능한 것을 나타낸다.



〈그림 11〉 MJFFS : 1,024kbyte 간격의 체크포인트에서 8,192kbyte 데이터 마운트

<표 3>은 체크포인트 간격을 1Mbyte, 2Mbyte, 4Mbyte, 8Mbyte 일 때 파일시스템에서 파일이 사용하는 용량과 체크포인트가 사용하는 용량을 계산한 것이고, kbyte 단위로 나타낸 것이다. 사용한 파일시스템은 16Mbyte의 플래시 메모리이며, 체크포인트는 하나의 파일에 대해 파일의 용량과 상관없이 1개의 체크포인트 블록으로 저장할 때의 값이다. <표 3>과 같이 체크포인트로 저장되는 정보의 크기는 체크포인트 간격과는 상관없이 거의 비슷하지만, 파일의 크기가 크면 클수록 체크포인트 정보는 작아진다. 파일시스템을 마운트하거나 오류 회복을 위해 접근하는 데이터의 양은 체크포인트 간격이 좁을수록 작아진다. 본 논문에서 구현하고자 하는 MJFFS의 체크포인트 그룹 간 간격을 1Mbyte로 하여 마운트의 효율성에 중점을 두었다.

〈표 3〉 파일의 크기에 따른 체크포인트 데이터

	1개 파일 크기 ≒ 10kbyte				1개 파일 크기 ≒ 20kbyte			
	1,024	2,048	4,096	8,192	1,024	2,048	4,096	8,192
전체 파일의 용량	15,616	15,608	15,604	15,604	15,984	15,984	15,984	15,984
전체 체크포인트 용량	768	776	780	780	400	400	400	400
파일의 크기가 8192kbyte일때 최대 마운트 시간	1,360	2,339	4,291	8,192	1,199	2,198	4,196	8,192

4.2 MJFFS의 성능

MJFFS는 로그 파일시스템(LSF - log structure d file system)이다. 파일의 내용이나 변경되는 로그 정보를 모두 로그 형태로 플래시에 기록한다. LSF의 장점은 시스템의 이상으로 인한 파일시스템의 오류를 순차적으로 처리함으로써 쉽게 파일시스템을 재구성 할 수 있다는 점이다. 플래시 파일시스템의 용량이 큰 경우 오류복구와 파일시스템 구성을 위한 순차적인 처리는 많은 시간을 요한다. MJFFS는 JFFS가 가지는 시간 지연에 따른 단점을 극복하기 위해 체크포인트를 사용한다. 체크포인트를 사용함으로써 파일시스템의 재구성 시간은 절반 이하로 줄어든다. 체크포인트는 파일시스템의 용량이 클수록 좋은 성능을 보인다.

MJFFS 파일시스템은 효과적인 오류 복구 기능을 가진다. 시스템의 전원이상으로 새롭게 파일시스템을 재구성하는 경우에 JFFS보다 빠른 시간에 오류 복구가 가능하다. MJFFS는 오류 복구 결과를 체크포인트를 사용하여 기록한다. 오류가 발생된 블록의 주소는 체크포인트에서 참조를 하지 않도록 하여 잘못된 정보를 참조하는 것을 방지하며 새롭게 추가된 노드에 대한 정보를 추가한다. MJFFS는 파일시스템의 재구성시에 체크포인트 정보를 갱신하고, 체크포인트 이후의 파일정보에 대해 오류가 없는 파일 정보를 추가함으로써 재구성이 완료된다. 이와는 반대로 JFFS는 파일시스템의 모든 파일을 검사하며 새롭게 재구성하는 방식으로 오류가 없는 파일에 대해서도 검사를 수행하기 때문에 그 효율이 나쁘다.

플래시 파일시스템은 플래시 메모리의 물리적인 주소와 논리적인 주소를 매핑(mapping)시키기 위한 메모리 매핑 테이블을 사용하고 있다. JFFS의 경우에도 마운트 과정을 통해 파일시스템의 모든 파일에 대한 정보를 메모리로 로드하고 그 정보를 이용하여 접근한다. MJFFS도 같은 방법으로 구현되지만, 사용하지 않는 정보를 모두 메모리로 로드하지 않는다. 체크포인트를 이용하여 정보를 수집하는데, 체크포인트는 그 파일에 대한 기본적인 정보만을 유지하고 있다. 체크포인트 정보 수집 후, 체크포인트로 기록되지 않은 파일에 대한 자세한 정보를 수집한다. 마운트가 끝나고 메모리 관리자가 유지하고 있는 정보는 대부분의 파일에 대해서는 간략한 정보만을 보유하고, 체크포인트로 기록할 파일만 자세한 정보를 유지한다. 파일시스템을 사용하면서 필요한 파일에 대한 자세한 정보를 추가적으로 획득하여 사용하기 때문에 MJFFS는 메모리의 사용량을 절약할 수 있다.

다음은 JFFS와 MJFFS의 마운트와 오류복구 성능을 비교한 것이다. 에뮬레이션에 사용된 플래시 메모리의 크기는 4Mbyte와 8Mbyte이며, 체크포인트 간격은 512kbyte 이다.

<표 4>는 8Mbyte의 플래시 메모리에서 10kbyte 파일이 각각 10~70%정도로 플래시 메모리를 사용중 일 때 마운트 시간을 측정한 것이다. 첫 번째 812kbyte를 사용하고 있는 플래시 메모리에서 JFFS가 마운트하기 위해서는 파일시스템 전체를 모두 읽어야 하지만, MJFFS의 경우에 최소로 소요되는 시간은 512kbyte의 간격을 가진 체크포인트 그룹 2개를 읽는 것이고, 최대 시간은 체크포인트 그룹 1개와 체크포인트 그룹으로 기록되지 않은 487kbyte를 읽는 시간이 된다. 실험에서 사용한 방법은 파일시스템의 처음부터 시작하여 플래시 메모리의 10%를 사용하는 시점에서 마운트 시간을 측정한 것이다. 이를 수식으로 나타내면 다음과 같다.

최소 소요 시간 : $T_{sum} = T_{flash_read} \times 25k \times 2$

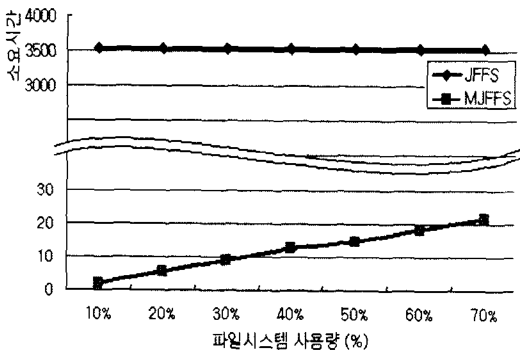
최대 소요 시간 : $T_{sum} = T_{flash_read} \times 25k \times 1 + T_{flash_read} \times 487k$

10~70% 사용량의 평균값을 보면, JFFS는 3523ms로 8Mbyte 플래시 메모리 전체를 탐색한 결과이고, MJFFS는 10ms의 평균값을 가진다.

〈표 4〉 8Mbyte의 플래시 사용량에 따른 성능

사용량	10%사용	30%사용	50%사용	70%사용
JFFS	3523ms	3523ms	3523ms	3523ms
MJFFS	2ms	9.1ms	14.7ms	21.7ms

<표 4>에 나타난 결과에는 메모리에서 처리하는 시간은 제외하고, 플래시 메모리에 대한 접근 시간만을 보여준다. 30%가 사용되는 8Mbyte 플래시 메모리에서 JFFS가 마운트하기 위해서는 8Mbyte를 읽어야 하지만 MJFFS는 127kbyte를 읽으면 가능하다. <그림 12>는 <표 4>의 결과를 나타낸 것이다. JFFS는 파일시스템의 사용량에 관계없이 동일한 결과를 보여주고, MJFFS는 파일시스템의 사용량에 대해 적은 변화량을 보여준다.



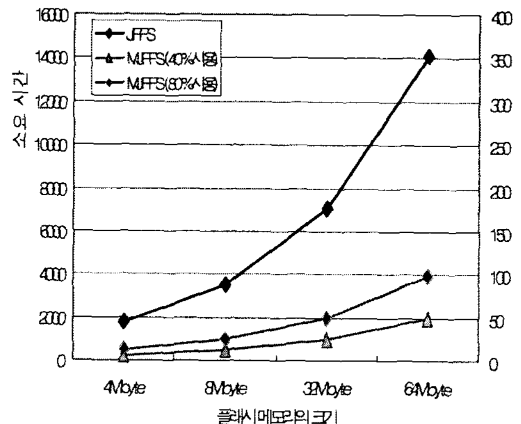
〈그림 12〉 JFFS와 MJFFS의 마운트 성능비교(사용량)

지금까지 8Mbyte의 플래시 메모리의 사용량에 따른 JFFS와 MJFFS의 마운트 성능을 비교해 보았다. 다음은 플래시 메모리를 같은 비율로 사용할 때 플래시 메모리의 크기에 따른 JFFS와 MJFFS의 성능을 비교해 본다. 기본적인 가정은 위와 동일하며 파일의 사용량이 40%인 플래시 메모리에 대해 비교한다. <표 5>는 파일시스템의 크기가 각각 4Mbyte, 8Mbyte, 16Mbyte, 그리고 32Mbyte에 대해 각각 40%, 80% 정도 사용한 결과를 보여준다.

〈표 5〉 용량에 따른 접근 성능 비교

		4Mbyte	8Mbyte	16Mbyte	32Mbyte
40% 사용	JFFS	1,761ms	3,523ms	7,046ms	14,093ms
	MJFFS	5.6ms	12.6ms	25.2ms	49.8ms
80% 사용	JFFS	1,761ms	3,523ms	7,046ms	14,093ms
	MJFFS	12.6ms	25.2ms	49.8ms	98.8ms

<표 5>의 결과를 그래프로 나타낸 것이 <그림 13>이다. 동일한 비율로 사용 중인 파일시스템에 대해 JFFS는 파일시스템의 크기에 비례하는 결과를 보여주고, MJFFS는 파일시스템의 크기의 변화에 비해 적은 변화량을 보여준다.



〈그림 13〉 JFFS와 MJFFS의 성능비교

(플래시 메모리 용량별 40% 사용)

5. 결론 및 추후과제

최근에 모바일 장비의 저장 매체로 전력소모가 낮고 부피가 작은 플래시 메모리의 사용량이 증가되고 있다. 더불어 플래시 메모리의 특성을 반영한 파일시스템이 다양하게 출시되고 있는데, 그 중 대표적인 것이 JFFS이다. JFFS는 로그 구조로 된 파일시스템으로 변경되는 파일의 내용을 수정하여 저장하는 것이 아니고, 변경 사항을 로그 형식으로 추가로 저장하고 READ 연산을 수행 할 때 변경된 사항을 적용한다. 이러한 순차적인 접근 방식으로 플래시 메모리를 효과적으로 사용하며 오류에 강한 파일시스템을 구성할 수 있다. 그러나 JFFS는 파일시스템의 사용 형태나 사용 중인 파일의 크기와 관계없이 동일한 마운트 시간을 갖는데, 이는 파일시스템 전체를 검사하여 처리하는 시간이다. 따라서 실제 사용 중인 용량이 작아도 불필요한 검사를 수행하여 지연시간이 오래 걸리게 되며, 플래시 메모리의 크기가 클수록 비례하여 증가된다.

본 논문에서는 플래시메모리를 이용하는 파일시스템인 JFFS를 개선한 MJFFS를 구현하고 성능을 평가하였다. MJFFS는 일정한 간격마다 체크포인트를 이용하여 파일에 대한 정보를 저장한다. JFFS를 이용한 파일시스템을 사용하면서 발생되는 처리시간과 메모리의 사용량, 오류회복 시간을 체크포인트를 이용하여 개선시켰다.

MJFFS는 모바일 장비의 파일시스템에 초점을 두어 연구되었다. 컴퓨팅 파워가 작고 이동성이 중요시되는 소규모 모바일 장비의 파일시스템에 적당할 것으로 보여 진다.

제안된 MJFFS는 전체 파일시스템의 한 부분이다. 향후 연구과제로는 캐시 메모리와 플래시 메모리를 효율적으로 사용하기 위해서 삭제할 공간과 가용 공간에 따른 cleaning 정책에 대한 연구가 지속적으로 필요하다.

참 고 문 헌

[1] 김정기, 박승민, 김채규, "임베디드 플래시 파일 시스템", 정보처리, 제 9권 1호, 2001년 1월, pp. 43-49.

[2] 박성호, 정기동, "내장형 운영체제의 파일 시스템 - Flash File System", 정보처리, 제 9권 3호, 2002년 5월, pp. 103-108.

[3] 김한준, 이상구, "신뢰성 있는 플래시 메모리 저장시스템 구축을 위한 플래시메모리 저장 공간 관리 방법," 정보과학회 논문집, 제 27권 6호, pp.567-582, 2000. 6.

[4] A. Kawaguchi, S. Nishioka, and H. Motoda, "A Flash-Memory Based File System," Proc. of the 1995 USENIX Technical Conference, Jan. 1995.

[5] M. Rosenblum and J.K. Ousterhout, "The Design and Implementation of a Log- Structured File System," ACM Transactions on Computer Systems, Vol.10, pp.26-52, 1992.

[6] David Woodhouse, "JFFS : The Journalling Flash File System", Technical Paper of RedHat inc. Oct. 2001.

[7] Intel Corporation, "3-Volt Advanced Boot Block Flash Memory, 28F160B3", Oct, 2001.

■ 저자소개

김 영 관

현재 (주)디지스타에서 PAN 네트워크 장비 소프트웨어 개발자로 일하고 있고, 한밭대학교 정보통신 대학원에서 공학석사(2004)를 취득했다. 주요 관심분야는 리눅스 임베디드 시스템, 파일시스템, PAN 네트워크 등이다.

박 현 주

현재 한밭대학교 정보통신컴퓨터공학부 조교수로 재직 중이다. 서울시립대학교 전산통계학사(1990), 서울대학교 계산통계학 석사(1992), 박사학위(1997)를 취득하였다. 주요 연구 분야는 파일시스템, 임베디드 소프트웨어, 모바일 데이터베이스(Mobile Database) 등이다.

◆ 이 논문은 2003년 8월 25일 접수하여 1차 수정을 거쳐 2004년 2월 25일 게재확정되었습니다.