과제 #6 : 파일 시스템

○ 과제 목표

- 가상 디스크(애뮬레이터)를 위한 간단한 파일 시스템 및 기본 파일 operation 구현
- 디스크 접근 기능이 구현된 ssufs-disk.h와 ssufs-disk.c에서 정의(구현된)된 여러 함수를 이용하여, ssufs-ops.c를 수정(ssufs_create(), ssufs_open(), ssufs_delete(), ssufs_read(), ssufs_write() 구현)

○ 기본 지식

- 가상 디스크
  - ✓ 64 바이트 크기의 35개 블록으로 구성
  - ✓ (참고) ssufs-disk.h와 ssusf-disk.c, 디스크 상에 간단한 텍스트 파일을 통해 디스크를 에뮬레이션
  - ✓ 첫 번째 블록 : 슈퍼블럭, 다른 모든 블록의 상태에 대한 요약
  - ✓ 다음 4 개 블록 : 블록 당 2 개의 inode가 있는 inode 구조체가 있음 -> 총 (최대) 8개의 파일
  - ✓ 나머지 30개 블록 : 데이터(파일) 블럭
  - ✓ 디렉토리는 별도로 없음 (루트 디렉토리만 있음)
- inode 구조체(32 바이트)의 구성 (ssufs-disk.h 39~49 라인 참고)
  - ✓ 파일 크기 (바이트)
  - ✓ 파일 이름
  - ✓ 최대 4 개의 파일 데이터 블록에 대한 직접 포인터 -> 한 파일 최대 크기는 256 바이트(4개의 데이터 블록)
- ssufs-disk.c에 구현된 함수 설명
  - ✓ int open_namei(char *filename) : 요청된 파일 이름이 filename인 inode 번호를 리턴. 에러 시(예. 이름이 filename인 파일이 없을 경우) -1 리턴
  - ✓ void ssufs_formatDisk() : 파일 시스템 포맷 및 슈퍼 블록 초기화
  - ✓ int ssufs_allocInode() : 디스크 상에서 free inode를 할당하고, 슈퍼 블록에 정의되어 있는 freelist에 있는 inode 갱신. 성공 시 파일에 할당된 inode 개수 리턴. 에러 시(예. free inode가 없을 경우) -1 리턴
  - ✓ void ssufs_freeInode(int inodenum) : 슈퍼블럭에서 inodenum으로 요청된 inode를 free로 지정
  - ✓ void ssufs_readInode(int inodenum, struct inode t *inodeptr) :디스크에서 inodenum으로 요청된 inode를 읽고 inodeptr에 복사
  - ✓ void ssufs_writeInode(int inodenum, struct inode t *inodeptr) : 요청된 inodeptr으로부터 inodenum 복사
  - ✓ int ssufs_allocDataBlock() : 디스크에 사용 가능한 데이터 블록을 할당. 슈퍼블럭에서 사용가능한 블록(freelist)를 갱신. 성공 시 할당 된 데이터 블록 수를 리턴. 에러 시(예. 가용한 블록이 없는 경우) -1 리턴
  - ✓ void ssufs_freeDataBlock(int blocknum) : 요청된 blocknum 블록을 슈퍼 블록에서 free로 지정
  - ✓ void ssufs_readDataBlock(int blocknum, char *buf) : 요청된 blocknum 블록을 읽어 buf에 복사
  - ✓ void ssufs_writeDataBlock(int blocknum, char *buf) : 요청된 blocknum 블록의 내용을 버퍼에서 읽어 디스크에 쓰기
  - ✓ void ssufs_dump() : 디스크의 상태를 프린트
  - ✓ 주의 사항

- 디스크에 모든 쓰기는 리턴 전에 에뮬레이트 된 디스크 (텍스트 파일)를 변경함
- 본 구현에서는 단 하나의 프로세스는 텍스트 파일을 변경(수정)한다고 가정. (두개 이상의 프로세스가 병행적으로 쓰기 작업은 지원하지 않음)
- 위에서 정의한 함수를 사용할 때는 에러를 확인하기 위해 함수의 리턴 값을 고려해야 함 -> 기타 에러는 별도로 고려하지 않고 구현해야 함. -> 에러를 리턴하지 않는 모든 디스크 관련 연산은 성공한 것으로 간주함 (예. 사용 가능한 데이터 블록이 없어 데이터 블럭을 할당하는 함수는 에러가 있을 수 있으나, 유효한 데이터 블록을 읽는 함수는 항상 성공적으로 리턴한다고 생각하고 구현해야 함)

○ 과제 기본 내용

- 기본지식에서 설명한 함수를 이용하여 과제 구현 내용의 기능 구현 (ssufs-ops.h 확인 및 ssufs-ops.c 수정)
- ssufs-disk.h : file handle 배열(ssufs-disk.c 4 라인 참고)이라고 하는 메모리 내 데이터 구조를 정의함 (ssufs-disk.h 51~57 라인 참고) -> Linux 및 Unix의 file descriptor table과 유사.
  - ✓ 시스템에서 오픈한 파일의 상태 (inode 번호, 읽기/쓰기 오프셋 등)는 오픈 파일의 file handle에서 유지됨
  - ✓ 파일시스템은 모든 오픈 파일의 file handle 배열을 유지하고 있어야 함
  - ✓ 이 배열의 file handle 인덱스는 오픈 후 읽기/쓰기 작업에서 오픈 파일을 고유하게 식별하는 용도로 사용
- Linux 및 Unix 시스템과 달리 create()와 open()은 별도로 구현 (open()에서 파일을 생성하는 기능 없음)
  - ✓ ssufs_create()은 디스크에 파일 생성. ssufs_open()은 이미 생성되어 있는 파일의 읽기 및 쓰기 모드 오픈

○ 과제 구현 내용

- 1. int ssufs_create(char *filename)
  - ✓ 파일 시스템에 요청된 filename으로 파일 생성
  - ✓ 파일 생성은 동일한 이름의 파일이 파일 시스템에 존재하지 않고 시스템에 추가로 파일을 생성할 공간이 있는 경우에만 성공해야 함
  - ✓ 해당 파일을 위해 사용 가능한 inode를 할당하고 디스크에서 적절하게 초기화 필요
  - ✓ 성공 시 : 새로 생성 된 파일의 inode 번호를 리턴
  - ✓ 에러 시 : -1 리턴
- 2. void ssufs_delete(char *filename)
  - ✓ 파일 시스템에서 요청된 filename으로 파일 삭제 (존재하는 경우에만)
  - ✓ inode와 같은 해당 파일과 관련된 리소스를 해제해야 함
  - ✓ close()를 하지 않아도 삭제해도 됨 (학생들이 자유롭게 구현)
- 3. int ssufs_open(char *filename)
  - ✓ 파일 열기
  - ✓ create()로 생성된 파일의 경우에만 성공
  - ✓ file handle 배열에서 사용하지 않는 파일 핸들의 한 항목을 할당하고 초기화한 다음 새로 할당된 file handle의 인덱스 리턴

✓ 에러 시 : -1 리턴
✓ 참고 : 파일의 읽기 및 쓰기는 본 함수로 리턴된 file handle 인덱스를 사용하여 파일의 inode 번호를 참조해야 함

- void ssufs_close(int file handle) - ssufs-ops.c에 구현되어 있음
✓ 열린 파일을 닫고 file handle 해제
✓ 주의할 점 : 디스크에서 파일을 삭제하지 않음
✓ 에러 시 : -1 리턴

- 4. int ssufs_read(int file handle, char *buf, int nbytes)
✓ open()된 파일의 현재 오프셋에서 요청된 buf로 요청된 nbytes 수를 읽음
✓ 요청된 nbytes 수는 디스크 상에서 연속되지 않을 수도 있음 -> 여러 블록에 걸쳐 있을 수 있음
✓ 성공 시 : 0 리턴
✓ 에러 시 : -1 리턴
✓ 주의할 점
- 파일의 일부 읽기는 지원 하지 않음. 요청된 nbytes 수를 모두 버퍼로 읽거나, 전혀 읽지 않아야 함
- 요청된 nbytes 수를 읽으면 파일 끝을 넘어 가게 되는 경우 아무 것도 읽지 않아야하며 에러를 리턴해야 함
- 읽기가 블록 경계에서 정렬되지 않고 블록 중간에서 시작하거나 끝날 수 있는 경우 반드시 처리해야 함
- 읽기가 성공하면 file handle의 오프셋 값도 갱신해야 함

- 5. int ssufs_write(int file handle, char *buf, int nbytes)
✓ open()된 파일의 현재 오프셋에서 요청된 buf에서 요청된 nbytes 수를 디스크에 씀
✓ 요청된 nbytes 수는 디스크 상에서 연속되지 않을 수도 있음 -> 여러 블록에 걸쳐있을 수 있음
✓ 성공 시 : 0 리턴
✓ 에러 시 : -1 리턴
✓ 주의할 점
- 파일의 일부 쓰기는 지원 하지 않음. 요청된 모든 nbytes 수를 파일에 쓰거나 전혀 쓰지 않아야 함
- 요청된 nbytes 수를 쓸 수없는 경우 (예 :요청 된 바이트 수를 쓰면 최대 파일 크기 제한을 초과하거나 쓰기를 완료하는 데 사용할 수 있는 free 디스크 블록이 없는 경우) 아무것도 쓰지 않고 에러 리턴해야 함.
- 쓰기가 실패한 경우 실패가 발생하기 전에 할당 된 모든 데이터 블록을 해제하고 해당 블록은 파일시스템에 반환해야 함. -> 실패한 쓰기는 파일시스템이 시작된 상태와 동일한 상태를 유지
- 쓰기가 블록 경계에서 정렬되지 않고 블록 중간에서 시작하거나 끝날 수 있는 경우 반드시 처리
- 쓰기가 성공하면 file handle의 오프셋 값도 갱신해야 함

- int ssufs_lseek (int file handle, int nseek) - ssufs-ops.c에 구현되어 있음
✓ file handle의 파일 오프셋을 요청된 nseek 바이트만큼 증가시킴
✓ Linux 파일시스템과 유사하게 오프셋은 파일에서 읽고 쓸 수 있는 다음 바이트를 나타냄
✓ nseek이 음수이면 오프셋이 감소해야 함
✓ ssufs_lseek()은 오프셋을 현재 파일 크기 경계 이상으로 이동시키면 안됨
✓ 성공 시 : 0 리턴
✓ 에러 시 : -1 리턴

- 기타 주의할 점

✓ 에뮬레이트 된 디스크 상의 디스크 데이터 블록에 대한 변경 사항을 구현하는 것임

- 구현 테스트

```
$ chmod a+x ssufs_test.sh
$ ./ssufs_test.sh
```

✓ 주어진 ssufs_test.sh를 활용하여 구현한 프로그램 기능 확인

✓ 주어진 ssufs_test.c 프로그램을 수정하여 다양한 테스트 케이스로 구현한 프로그램 기능 확인

✓ 구현을 완료 한 후에는 구현한 프로그램의 출력이 예상 출력(ssufs_test.output)과 정확히 일치하는지 확인

○ 과제 제출물 및 마감

- 2020년 12월 6일 (일) 23시 59분 59초까지 제출
- 제출물

✓ 개요 또는 구현 내용 (수정한 부분 간단하게 설명)과 실행 결과 캡쳐 등 자율적으로 보고서 작성

✓ 제출할 tgz(또는 zip) 내에는 보고서 파일과 위 세부 과제 1,2,3,4,5의 구현이 들어간 ssufs-ops.c 포함 (한 디렉토리 내에 보고서 파일과 ssufs-ops.c파일이 존재하면 됨)

- 별도 테스트 프로그램은 보고서에 포함시켜도 되나 제출하지 말 것

- 마감 : 2020년 12월 6일 (일) 23시 59분 59초까지 구글클래스룸으로 제출

○ 필수 구현

- 1

○ 배점 기준

- 1 : 10점
- 2 : 25점
- 3 : 15점
- 4 : 25점
- 5 : 25점