# CS 445/445G Project 1: A Simple File System

**(due at the midnight on 04/07/24)**

In this project, you will implement a simple simulated file system. This project will let you get familiar with important data structures in a file system, including i) volume control block, ii) the directory structure, iii) open file tables (system-wide and per-process), and iv) File Control Block (FCB). You will also learn how basic file operations, such as open(), interact with these data structures.

Before we describe the tasks of this project, we make the following assumptions. The simulated file system is implemented on main memory. More specifically, a slot of main memory of 1M is used to simulate a disk. We assume that a data block is of 2K. Therefore, the disk has 512 blocks. In this project, we assume the disk is a data disk, i.e., no OS installed. So the *first* data block is used as the volume control block. We assume that the volume control block contains the following items:

| number of blocks |
|---|
| size of block |
| a free-block count |
| a bit-map of free blocks |

Table 1: Volume control block

We assume that the file allocation (i.e., data block allocation) is **contiguous allocation**. For simplicity, our file system uses a flat directory structure to manage files. In other words, the directory contains no subdirectories. With the assumption of contiguous allocations, the content of the directory of our file system can be organized as a table as follows:

| file_name | start_block_number | file_size |
|---|---|---|
| file1 | 0 | 2 |
| file3 | 6 | 3 |
| ⋮ | ⋮ | ⋮ |

Table 2: Flat directory structure.

To be consistent with the contiguous allocation, the FCB contains the following items:

| file size |
|---|
| pointer to the first data block |

Table 3: File control block (FCB).

A system-wide open file table is a set of FCBs of open files. An example of a system-wide open file table is shown in Table 4.

Given a specific process, its per-process open file table contains a set of file handles of files opened by the process. For example, assume that a process has open files: f1 and f4, its per-process table will be the one shown in Table 5.

| file_name | FCB |
|---|---|
| f1 | FCB1 |
| f3 | FCB3 |
| f4 | FCB4 |

Table 4: System-wide open file table.

| file_name | handle |
|---|---|
| f1 | 0 |
| f4 | 2 |

Table 5: Per-process open file table.

# 1 Part I: Basic file operations

You will need to implement the following file operations:

- create(). create a file with a specified size. The size is in terms of the number of blocks.
- open(). open a file, and update system-wide and per-process open file tables.
- close(). close a file, and update system-wide and per-process open file tables.
- read(). read a file to a local variable.
- write(). Write specified content to selected free blocks.

**CS 445G** graduate students should implement two additional file/directory operations:

- dir(). display all files in the file system.
- delete(). remove a specified file from the directory.

# 2 Part II: A simulation

Write an application to demonstrate a sequence of file operations of different *threads*. A better way of simulation is based on processes rather than threads. (Recall that threads in the same process share the same open files.) But in this project, we use a thread to simulate a process to simplify your work in case that you use Java language for the implementation. The following is a sample of file operations, where p1, p2 and p3 are three pthreads. p2 and p3 should be created simultaneously after p1 is completed.

1. p1: create file1
2. P1: write file1
3. p1: close file1
4. p1: create file2

5. P1: write file2
6. p1: close file2
7. p2: open file1
8. p2: read file1 and print it in the screen
9. p2: close file1
10. p3: open file2
11. p3: read file2 and print it in the screen
12. p3: close file2

**CS 445G** graduate students should further test the correctness of dir() and delete() operations. For example, you can insert a dir() call in front of the above simulation and append a delete() and a dir() at the end of the above simulation.

This project is a team project, and each team can have up to 2 students. You can use C/C++ or Java to implement this project. But if you use Java, keep in mind to demonstrate the simulation with multi-threading Java programming.

**On submission:**
Submit your source code and readme to blackboard. The source code should be well-documented, i.e., having good readability. The readme file should summarize: i) tasks that you completed, and ii) tasks that you did not complete.

# Appendix: Pthread

If you are going to use C, you can read the following tutorial on the use of pthread library.
**How to compile:**

```
> cc file.c -lpthread
```

The above compilation command means the user code should be linked to pthread.lib. Without this linking, the compiler will complain "undefined reference to" pthread functions, such as pthread_create, pthread_join.

**Sample code:**

```
#include<pthread.h>
#include<stdio.h>
```

```
int sum;     /* this data is shared by the threads */
void *runner(void *param); /* the thread */

int main(int argc, char *argv[])
{
  pthread_t tid;    /* the thread identifier */
  pthread_attr_t attr;  /* set of thread attributes */


  if (argc != 2)
  {
     fprintf(stderr, "usage: ./a.out <positive integer>\n");
     exit(0);
  }

  /* get default attribute */
  pthread_attr_init(&attr);

  /*create the thread */
  pthread_create(&tid, &attr, runner, argv[1]);

 printf("Main thread is busy doing something ...\n");
  while (sum <= 1)
  {
      printf("%d ", sum);
  }


  /* wait for the thread to exit */
  pthread_join(tid, NULL);

  printf("sum = %d \n", sum);
}

void *runner(void *param)
{
  int i, upper = atoi(param);
  sum = 0;

  for(i = 1; i <= upper; i++)
  {
    sum += i;
    //    printf("%d ", sum);
  }
```

```
  pthread_exit(0);
}
```