Programming Assignment 1

February 5, 2007

Goal

The purpose of this assignment is to get us up to speed with the basics of Unix C programming. The first part is about file input and output (disk emulator), while the second part is about memory management and data structures (LRU cache).

Part 1: Disk Emulator

Overview

The goal here is to read and write blocks of data to and from a file.

There are 3 basic function that need to be implemented for the disk emulator.

The first is

int init_disk(char *filename, int block_size, int num_blocks)

The arguments are:

char *filename: the name of the file that should be used to store the data

int block_size: the size of each block

int num_blocks: the number of blocks

Return value:

On success returns 0

On failure returns 1 (because the file couldn't be created, etc.)

Function description:

You should create a file called filename that is of size block_size*num_blocks. I recommend you fill it with zeros up to this size and check that everything is successfull. That way you avoid having the program crash later because you've run out of space.

Also since every other function depends on block_size and num_blocks, you should probably save them in global variables.

The second is:

int read_blocks(int start_address, int nblocks, void *buffer)

The arguments are:

int start_address: the address of the first block to be read

int nblocks: the number of blocks to be read (starting from start_address)

void *buffer: where to store the data that is read from the disk

Return value:

Returns the number of blocks that were read.

Function description:

You should seek to the beginning of block number start_address, and then read nblocks blocks and copy them into buffer.

Error conditions:

You should make sure that the start_address and nblocks are within the proper limits.

There are a number iof additional requirements:

Reading a block should fail with a small probability p There should be an additional delay of L milliseconds when reading a block.

The third is:

int write_blocks(int start_address, int nblocks, void *buffer)

The arguments are:

int start_address: the address of the first block to be written

int nblocks: the number of blocks to be written (starting from start_address)

void *buffer: the data to be written to the disk

Return value:

Returns the number of blocks that were written.

Function description:

You should seek to the beginning of block number start_address, and then write nblocks blocks from buffer to the disk.

Error conditions:

You should make sure that the start_address and nblocks are within the proper limits.

There are a number iof additional requirements:

Writing a block should fail with a small probability p There should be an additional delay of L milliseconds when writing a block.

Part 2: Cache

Overview:

Once you've finished the 3 disk emulator functions then you can add the cache. The cache is an in-memory data structure used to speed up reads and writes.

You will need to modify read_blocks and write_blocks to use the cache.

read_blocks should check that the data it wants is in the cache and if so it should read it directly from the cache, if not it should copy the data into the cache first. Similarly write_blocks should check to see if the data is already in the cache, if it is then it should update it, if not it should store it in a new block of cache.

However if the cache is already full then some blocks may need to be replaced. You should use a least-recently-used (LRU) replacement policy. Remember to write modified data back to the disk.

The cache will also require 2 additional functions.

To initialize it:

int init_cache(int cache_size_in_blocks)

The arguments are:

int cache_size_in_blocks: literally the cache size in blocks

Return value:

0 on success

1 on failure (e.g. couldn't allocate enough memory)

Function description:

This function initializes a cache of the proper size. This is where you should allocate any memory you will need for the cache using malloc(). Since this function can also be used to resize the cache you can use free() and realloc() as needed.

Error conditions:

You should check that cache_size_in_blocks is a valid number.

To flush it:

int flush_cache()

Return Value:

0 on success

1 on failure (e.g. disk full)

Function description:

You should write all dirty blocks back to the disk. That is to say any data written to the cache but not yet saved on disk should be saved to the disk.