Assignment 3: RAID Simulator

- Due Apr 3 by 10p.m.
- Points 1

Assignment FAQ and updates: A3 Raid Sim FAQ

In this assignment you will implement part of a RAID system. RAID stands for Redundant Array of Inexpensive (or Independent) Disks, and was invented by Garth Gibson and his PhD supervisors David Patterson and Randy Katz at UC Berkeley. Garth Gibson is Canadian and was the first president and CEO of Vector Institute here in Toronto. The key idea behind RAID is to provide fault tolerance for data stored on hard drives, and improve performance when reading data from the storage system. There are multiple versions of RAID, known as levels, that provide different types of fault tolerance and have different performance characteristics. For this assignment we will be implementing a variation of RAID level 4 where data blocks are striped across the disks in the system and there is one disk that stores the parity blocks computed for each stripe.

The file system interface to a storage device is logically a sequential array of blocks. The file system is made up of data structures that map file names to the blocks that store the contents of the files. The structure and implementation of file systems is a major topic in CSC369, so for this assignment we are only going to work at the RAID controller level and will only read and write blocks without consideration of the files that they might belong to.

In the diagram below, each hard drive stores a sequence of blocks. Blocks are striped across the disks in the system and a parity disk is used so that if a disk fails, the data on the failed disk can be reconstructed. Each disk holds `disk_size` bytes (or `disk_size/block_size` number of blocks). You can see that the logical ordering of the blocks is across the stripes, so logical block 0 is stored on disk 0, logical block 1 is stored on disk 1, and so on.

RAID_diagram.pdf

Download RAID_diagram.pdf

1. Overview

In this simulation, the main program (in `raid_sim.c`) will perform operations at the block level on the RAID system. There are two ways to enter commands: a shell-like interface where you can type command from the keyboard, and a transaction file interface similar to what you saw in A2. The commands that can be sent to the RAID system are:

- `wb <block_num> <filename>`

    - Read the first `block_size` number of bytes from a regular file named `filename`, and write that data to block number `block_num`
    - This gives us an easy way to write a block to the RAID system. The data to write is read from a local file.
- `rb <block_num>`
    - Read block number `block_num` from the RAID system and print it to `stdout`.
    - Although this simulation is implemented so that any type of data can be stored in the simulated disk, you may want to stick to ASCII characters to make testing easier.
- `kill <disk_num>`
    - Simulate disk failure by sending the `SIGINT` signal to the disk process with id `disk_num`
- `exit`
    - Send a checkpoint command to each disk, and then `wait` for the child processes to terminate.
    - The checkpoint command tells a disk to write its data to a file and then call exit.

Read the code in `raid_sim.c` to see how this code works. You do not need to change `raid_sim.c`. There are several helper functions that carry out these commands, and eventually make controller interface calls. (You may find it useful to draw a diagram showing the flow of control from the RAID commands described in this section to the controller functions that generate disk commands that are sent to the child processes.

While you are reading `raid_sim.c`, notice that there are three global variables that serve as configuration parameters. They have default values provided by defined constants, but can also be given different values using the command line arguments.

2. Controller

The `controller.c` file contains both the functions to initialize the disks in the RAID system and the functions that serve as the interface to the RAID system. Most of your work will be done in this file. The `read_block` and `write_block` functions implement the RAID operations, including updating the parity data. These functions are described below and in the starter code.

```
int init_all_controllers(int num_disks);
```

This calls `init_disk` for each of the disks in the system including the parity disk. These two functions implement all of the setup of the pipes and child processes. Remember that it is important to close unused pipe descriptors.

In this simulation, each disk in the RAID system is represented by a child process. Two pipes are are created for each child process, one for the parent to send commands and data to the child process, and one for the parent to read data sent by the child process. The `controllers` array is used by the parent process to store the pid of each child process and the pipe descriptors used to communicate to each child. The index into the controllers array corresponds to the disk id.

After all the pipes are set up properly, the `init_disk` function calls `start_disk` which serves as the main function for a child process and is implemented in `sim_disk.c`.

To interact with a disk process (child process), we need to send messages (or commands) through the pipe that it can interpret and execute. The next three functions will send one or more commands to the pipe and may also wait for results.

`char *read_block(int block_num, char *data);`

Reading is the simplest operation in this RAID system because all we need to do is read the appropriate block. This function calls `read_block_from_disk` which carries out the communication between the controller (parent process) and the disk (child process). This function should check that `block_num` is valid, so that you don't need to check for this error in the child process.

Note that there is an update to the comment for this function and the next one in the FAQ, linked at the top of this document.

`int write_block(int block_num, char *data);`

Writing a block to the RAID system means that we need to update the parity block. For example, if we want to write block `k'` to a RAID system with `N` disks, there are a few steps that we need to take:

1. Read block `k`
2. Read the parity block for the stripe that block `k` belongs to: `stripe number = k / N` (integer division)
3. XOR block `k` with the parity block, to cancel out its presence in the computed parity
4. XOR the new block, `k'` with the parity block.
5. Write the revised parity block to the parity disk
6. Write block `k'` to the appropriate disk

This function calls `read_block_from_disk` and `write_block_to_disk`, to carry out the communication between the controller and the disk. `write_block` should also check that `block_num` is valid, so that you don't need to check for this error in the child process.

`void checkpoint_and_wait();`

This function is called when an exit command is sent. This function is provided for you to illustrate how to send a command to the disk.

`void simulate_disk_failure(int disk_num);`

This function simply sends the SIGINT signal to the `disk_num` process to terminate it.

`void ignore_sigpipe();`

When you write to a pipe that does not have any readers, a SIGPIPE signal is sent. The default behaviour of a SIGPIPE is to terminate the program. The provided `ignore_sigpipe` function overrides this signal handler, ignoring the error instead. We want to ignore this error in order to recover from the case when a disk process is killed.

`void restore_disk_process(int disk_num);`

This function first calls `restart_disk` to start a new child process and then uses the parity disk and the data from the other disks to reconstruct the data for the disk that failed. This function should only be called when we detect that a disk has failed. We can tell when a disk has failed because a write call will return -1. You should add the check for a write call returning -1 to `read_block_from_disk` and `write_block_to_disk`. When you see that a write call returns -1 in these functions, call `restore_disk_process`. Note that the command that triggers the restore should have no effect beyond restoring the disk.

3. Disk commands

The `start_disk` function first allocates memory to store the data for the disk. Since we are using small disk sizes for the simulation, it is safe to allocated this data memory on the stack. Remember to initialize the disk data to 0. After this setup, the disk process will enter an infinite loop where it reads and handles commands. Because it needs to know what to expect to read from the pipe, it first reads a "command" from the pipe. A command is defined as an enum, and is an int that can have one of three values: `CMD_READ`, `CMD_WRITE`, and `CMD_EXIT`.

Once the command has been identified, the process knows whether it need to read additional arguments from the parent process. Each of the three commands has a different sequence of operations:

- `CMD_READ`:
  - read the block number from the parent
  - write the block to the parent. Since the disk data is stored as an array of bytes, we can calculate where the block starts and ends in this array and pass the appropriate address to the write call.

- `CMD_WRITE`:
    - read the block number from the parent
    - read the block data from the parent
    - ensure that the block data is stored in the correct location in the disk data array. (You can read it into the correct location directly or copy it there after the read call.)
- `CMD_EXIT`:
    - call `checkpoint_disk`, free any memory that you may have allocated, and exit.

Recommended approach

It is highly recommended that you spend some time reading through the code and following the path of each operation from the transaction file interface through to the communication with a disk process. Because there are several pieces to this system, you will want to have a thorough understanding of the starter code before you write any code yourself.

I would recommend that you start with the path that writes one block to a data disk, followed by the code to read one block and print it to stdout. Once you have that working you can add the code to update the parity block. Then you should test this code carefully to make sure that it handles valid blocks across different stripes with varying numbers of disks. Once you are satisfied that this part of the code is working well, only then should you move on to the functions that simulate a disk crash and restoring a disk.

Example:

I have sprinkled through my code debugging statements that help me understand what the code is doing. So you may write different debugging statements than the output here. In this simplest example, we are running raid_sim with 1 data disk, a block size of 16, and a disk size of 16 blocks. The contents of `data1`, `data2` and `simpletrans` are shown below.

```
$ cat data1
aaaaaaaaaaaaaaaa
$ cat data2
bbbbbbbbbbbbbbbb
$ cat simpletrans
wb 0 data1
wb 1 data2
exit
$  ./raid_sim -n 1 -t simpletrans
[0] Waiting for command
wb
[1] Waiting for command
[0] cmd: 0. Block num: 0, size: 16
```

```
[0] Writing data to parent. Block num: 0
[0] Waiting for command
[1] cmd: 0. Block num: 0, size: 16
[1] Writing data to parent. Block num: 0
[1] Waiting for command
[1] Read block from pipe. Block num: 0, size: 16
[1] Writing data to disk. Block num: 0, size: 16
[1] Waiting for command
Block 0 written to RAID
wb
[0] Read block from pipe. Block num: 0, size: 16
[0] Writing data to disk. Block num: 0, size: 16
[0] Waiting for command
[0] cmd: 0. Block num: 1, size: 16
[0] Writing data to parent. Block num: 1
[0] Waiting for command
[1] cmd: 0. Block num: 1, size: 16
[1] Writing data to parent. Block num: 1
[1] Waiting for command
[1] Read block from pipe. Block num: 1, size: 16
[1] Writing data to disk. Block num: 1, size: 16
Block 1 written to RAID
[0] Read block from pipe. Block num: 1, size: 16
[1] Waiting for command
[0] Writing data to disk. Block num: 1, size: 16
[0] Waiting for command
[1] Checkpointed disk
[0] Checkpointed disk
$ xxd disk_0.dat
00000000: 6161 6161 6161 6161 6161 6161 6161 6161  aaaaaaaaaaaaaaaa
00000010: 6262 6262 6262 6262 6262 6262 6262 6262  bbbbbbbbbbbbbbbb
00000020: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000030: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000040: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000050: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000060: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000070: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000080: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000090: 0000 0000 0000 0000 0000 0000 0000 0000  ................
000000a0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
000000b0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
000000c0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
000000d0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
000000e0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
```

```
000000f0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
$ xxd disk_1.dat
00000000: 6161 6161 6161 6161 6161 6161 6161 6161  aaaaaaaaaaaaaaaa
00000010: 6262 6262 6262 6262 6262 6262 6262 6262  bbbbbbbbbbbbbbbb
00000020: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000030: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000040: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000050: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000060: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000070: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000080: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000090: 0000 0000 0000 0000 0000 0000 0000 0000  ................
000000a0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
000000b0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
000000c0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
000000d0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
000000e0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
000000f0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
```

Style and Error checking

Similar to A2, TAs will be reading your program . You will be marked for programming style and error checking. All system calls should be checked for errors, using perror where appropriate.

You may assume that the transaction files are correctly formatted. The code to parse these transactions is given to you in raid_sim.c.

What to submit

Commit to the repository all code needed to compile and run your program. You are welcome to include transaction and data files that you use to test your program, but for this assignment, you do **not** need to submit test cases. Remember not to submit object files or executable files.