Tutorial 10: Using the filesystem

http://www.codemii.com/2009/03/02/tutorial-10-using-the-filesystem/

It’s been a while since I put up a tutorial and I hope that by now you have a basic understanding of programming for the Wii. So that I’m able to keep producing these tutorials I’ll try to focus on small pieces of code instead of the whole application and how the code can be applied. The explanation of Simon was a bit too long ;). This tutorial is also a bit longer than I expected.

In this tutorial we’ll cover initialising the filesystem, creating/deleting directories/files and reading/writing files. Thanks to joedj for his FAT init code from FTPii v0.0.5.

Note: This tutorial is for DevKitPPC r15. Users of r16 will have to prepend SD:/ whenever accessing the filesystem.

Initialising the filesystem

First things first, before we can start reading/writing files or creating/deleting directories we need to initialise the filesystem (FAT in this case). It’s just a simple piece of code which does this for us.

```
if (!fatInitDefault()) {
   printf("Unable to initialise FAT subsystem, exiting.\n");
   exit(0);
}
```

The above just reads, if we can’t initialise the FAT filesystem, then print an error message and exit.

The next thing to check (which isn’t mandatory) is to see if we can open the filesystem which is done with the code below.

```
if (!can_open_root_fs()) {
   printf("Unable to open root filesystem, exiting.\n");
   exit(0);
}

bool can_open_root_fs() {
   DIR_ITER *root = diropen("/");
   if (root) {
      dirclose(root);
      return true;
   }
   return false;
}
```

We use diropen to check if we are able to open the root directory and if so it returns true.

Ok, so we can open the filesystem. We can add another check to make sure that we can actually change the current working directory to the root (/):

```
if (chdir("/")) {
   printf("Could not change to root directory, exiting.\n");
   exit(0);
}
```

And now we're done with the initialisation. We can place this all in a function to make things a bit cleaner. We call our own die() function if something has failed.

It's always good practise to unmount the filesystem when exiting your application. This can be done by calling fatUnmount(0);.

```
void initialise_fat() {
   if (!fatInitDefault()) die("Unable to initialise FAT subsystem, exiting.\n");
   if (!can_open_root_fs()) die("Unable to open root filesystem, exiting.\n");
   if (chdir("/")) die("Could not change to root directory, exiting.\n");
}

bool can_open_root_fs() {
   DIR_ITER *root = diropen("/");
   if (root) {
      dirclose(root);
      return true;
   }
   return false;
}

void die(char *msg) {
   perror(msg);
   sleep(5);
   fatUnmount(0);
   exit(0);
}
```

As you can see above the die function accepts a string of characters (denoted by char *) and then prints out the string using perror(). The thing about perror is it will print out the string supplied as well as any error that has occurred. For example if you are trying to open a file and it doesn't exist, perror might print something like "No such file or directory" which is always helpful.

After that we just pause for 5 seconds, unmount the filesystem and exit the application.

Creating Directories

We can create a directory by using mkdir. mkdir takes two arguments, the directory to create and the permissions to set on the directory. We can ignore the permissions and just set them as 0777 (everyone has access to read, write and delete).

```
mkdir("/test", 0777);
```

The above would create a directory called /test on the root of the device we are accessing. mkdir returns 0 if the directory was created successfully otherwise it returns -1.

We can check for errors by doing the following:

```
if (mkdir("/test", 0777) == -1) {
   die("Could not create /test directory.\n");
}
```

This will run the mkdir command and if it returns -1 it will print the string specified and perror() will print out the error message.

Removing Directories/Files

We can remove a directory or a file by using unlink(path);

```
unlink("/test/myfile.txt");
unlink("/test");
```

The above would remove the file called myfile.txt and then the directory /test.

We can also check for errors exactly the same as mkdir:

```
if (unlink("/test") == -1) {
   die("Could not delete directory/file.\n");
}
```

Reading Files

There are various ways to read files. We'll stick to the easiest way which is reading the file one line at time so that we may process each line. This might be useful say if you're developing a game that has multiple levels and each line might represent an object's x and y co-ordinates.

First we need to open the file in read mode as below.

```
FILE *f = fopen ("myfile.txt", "rb");
```

Next we check if the file was able to be opened for reading.

```
// If file doesn't exist or can't open it then we can grab the latest file
if (f == NULL) {
   die("Could not open myfile.txt file for reading.\n");
} else { // Otherwise if it was fine, we can continue.
   char file_line [20];
   int line_number = 0;
   while (fgets (file_line, 20, f)) {
      printf("Line %i: %s\n",line_number, file_line);
      line_number++;
   }
   fclose(f);
}
```

We need store each line to a variable so that we are able to process the line. This variable is file_line and is an array of characters which holds 20 characters (I'm assuming that our x and y co-ordinates are between 1 to 3 characters in length, so 20 is more than enough). If say you had a long line of text, you would have to change 20 to say 500 or more depending on how long each line was. After that we just have the line_number which just counts the number of lines.

Now we reach the while loop, here we use the fgets function. fgets allows us to specify the amount of characters to read in the line or until the line ends. The first argument is a pointer to our array of characters which is where the information from the file will temporarily be stored. The second argument is the number of characters to read, again we use 20. The third argument is the pointer to the file, which is f in this case.

In every loop, file_line will be updated with the next line from the myfile.txt file. This continues until we reach the end of the file. We then print out the line number and the contents of the line at each loop.

After we have reached the end of the file, we can close the file by using fclose(f);

So let's say that our file contains the following data which represents x and y co-ordinates: 32 58 34 48 234 99 385 234 453 347

We need a place to store the x and y co-ordinates, a simple example is:

```
int x_pos[] = { 0, 0, 0, 0, 0};
int y_pos[] = { 0, 0, 0, 0, 0};
```

We would need to modify our while loop to break up each line, as we need the x and y co-ordinates in different variables.

Here we can use the string tokenizer (strok) to break up the line. strtok takes 2 arguments, the string to tokenize and a list of delimiters; which just means the characters to find that will determine a split in the string.

For our example the delimiters would just be a space (" ") which would break the line into 2 parts. The delimiter isn't included in any of the parts.

Now if we where to store each part directory into the integer variable the compiler would complain (char being stored in an int), so we need to change the string of characters to an int. This can be done using atoi(string) which just converts the string into an integer.

```
while (fgets (file_line, 20, f)) {
   char *temp_string;
   temp_string = strtok (file_line, " ");
   if (temp_string != NULL) {
      x_pos[line_number] = atoi(temp_string);
      temp_string = strtok (NULL, " ");
      if (temp_string != NULL) {
         y_pos[line_number] = atoi(temp_string);
      }
   }
   line_number++;
}
```

We firstly assign a blank pointer. We run strtok on our file_line variable which contains the line of text.

temp_string would then be equal to the first token in the line of text, which is “32″. Since temp_string has a value, we assign element 0 of x_pos with the integer value of 32.

We run strtok again, this time with the first argument of NULL. You use the NULL argument in strtok if you wish to continue breaking up a string.

The next token is “58″ which we assign the integer value of 58 to element 0 of y_pos. We then increase the line_number, and so on.

Writing files

As with reading files, writing files can be done in different ways. A simple way of writing files would use the fputs function which takes 2 arguments, the string to write to the file and the pointer to the file.

```
FILE *f = fopen ("myfile.txt", "wb");
if (f == NULL) {
   die("Could not open myfile.txt file for writing.\n");
} else {
   int x;
   for (x = 0; x < line_number; x++) {
      char temp_string[20];
      sprintf(temp_string, "%i %i", x_pos[x], y_pos[x]);
      fputs (temp_string ,f);
   }
   fclose(f);
}
```

Note that “rb” has changed to “wb” (the w means write).

We’ll use the same example as in the reading example, except in this case we wish to write our x and y co-ordinates to the file. We’ll assume that you have the line_number variable and that it equals 5. We do a for loop until we reach line_number (5).

So we need to do the opposite of strtok, so that we are able to add the two numbers and a space in between. We also need to change the integer values of x and y into strings.

We can do this using sprintf which has 3 arguments and is just like printf except it prints the output to a string. The arguments are the string to output to, the text to print and the variables to print.

We have the temp_string variable which is empty, we do a sprintf which converts “%i %i” to “32 58″. After we convert the integers to a string, we write this temp_string to our file and repeat until we are done.

Note: I have experienced issues when using sprintf multiple times in a row which can cause issues with the variables sprintf is writing to.

And that’s it; we’re done for this tutorial. You should now know how to preform different functions on files/folders which should make your life easier if you are say making a multi-level game or wish to save some simple settings to a file. Remember that there is a lot more error checking that could be done but for the purposes of this tutorial it’s just too much to do.

See you next time…