Code Write-up

Files: parallel_convert.c, Makefile, and time.sh

The overall design decision for this project is as follows:

We decided to keep all of our code in one C file. We felt that the program wouldn't really benefit from being abstracted out into multiple files. We decided that using multiple functions would keep the code from becoming too complex.

The functions:
int convertFile(long process, char* file, char* file_in, char* file_out, FILE *bad, FILE *log);
This function is used to handle the comparison of process ids and file extensions, and then handles conversion if the criteria is met. It made sense to create a function to handle this due to the many comparisons needed. The return value is an int. Returning 0 means the process completed a task successfully, 1 means that the process id was invalid for the current file.

int getNextFile(int* fileDone, int total_files);
This function is used to handle checking the fileDone array. It returns the index of the next unprocessed file. If there isn't an unprocessed file, it checks for running processes, and returns -1, which tells the main process that we need to wait for processes to return before we can determine the next action. The last check of the function is if all files have been processed, in this case -2 is returned to the main program, letting it know that file conversion is done.

The main function handles reading in the files, process creation, waiting, and html creation. The more complex part of our program revolves around the process creation and handling, so I will describe the structure that we used to handle this part:

pid_t pids[max_process]; An array of the pids being run. This array is used to determine which pid is being run, and then checked to see which process returns after wait calls. Will be as long as the max number of processes, input by the user as convert_count.
int processFile[max_process]; An array of the file index being processed by a specific process. As an example, processFile[0] = 12, would tell me that process 0 is currently working on file 12. Will be as long as the max number of processes, input by the user as convert_count.
int fileDone[total_files]; An array that contains the status of each file we need to process.
0 - file is unprocessed, 1 - file is processed successfully, 2 - file is currently being processed.

Using these 3 arrays we are able to manage all files and make sure that at any given time only 1 child process is working on any specific file. This prevents duplicate files from being converted, and allows us to always have convert_count processes running in parallel.
As an example, if the main program is waiting for a process, we get a pid back from wait, we then loop through and compare this pid to the pids[] array, the matching index i is then used to index in to processFile[i] and give us the current_file index that that process was working on. If

the status returned from wait was a success then we can sent the fileDone[current_file] index to 1 marking the file to complete, then create a new process in pids[i] and give that new process a new unprocessed file that we get from calling the getNextFile function. If the wait had returned an status of failure then we know that the process could not handle that specific file, thus we simply create a new process in pids[i] with the same file in hopes that this new pid can handle the file.

The general layout of the rest of main() is as follows:
Line 38-71: Variable declaration.
Line 75-85: Check the command line arguments to make sure that they are valid.
Line 95-102: Check for output directory and create it if it doesn't exist.
Line 105-152: Read the files in the input directory, build the correct path strings, then initialize the file arrays with the file strings.
Line 155-165: Initialize the the fileDone and processFile arrays with the correct initial values.
Line 167-190: Open the log file and nonImage file for writing.
Line 200-288: Process creation, calls to convertFile, and waiting until all files have been processed.
Line 200-288 is where we used the arrays that I described in detail above. We used a doubly nested while loop to handle process creation and waits. The first while loop runs until all files have been processed successfully and the nested while loop creates processes while there are available processes. Within the main while loop we handle the waiting for child processes. If a child process returns then we increment available processes and the nested while loop will then create a new process. This continues on until all files are done.

Line 310-355: Read the output directory and get the names for the HTML files, store in an array.
Line 357-395: Actually builds the HTML files. Loops through and concats the correct html to the html file, and then links it to the next html file name from the array in the previous step.

That is a general overview of our program. We felt that our code flowed nicely. The most complicated part is the process creation and handling, and we felt that our decision to use three main arrays really helped keep the indexing straight forward and easy to follow. The way we structured our program also allows for a variable number of file types to be used, which we thought was a nice addition.

For a more in depth explanation on specific code used, refer to comments throughout the program.