Project Specifications

- A number of potential functions had to be removed in lieu of brevity, as well as in lieu of optimisation. As a result, we removed a few functions (echo, syspath) and added some new ones (print). This decision was made in order to ensure that users using filewatcher were getting exactly their requirements for this application (by adhering to the user stories) - while also keeping in mind the constraint of time.
- We decided to rework our queue for a recursive child-fork process. This sacrifices our potential for queueing commands, but, when considering the purpose of the filewatcher, we decided that having multiple children to each handle their own process, and growing when required, was both the easier and more effective solution.
- Against our user story, we decided that users using filewatch would like to have their function retained, even after system crashes. This is because of the requirements; we would rather have a function keep monitoring, despite the outer shell breaking. This increases the robustness of our system.

Module 1

Window Dressing:

Despite previously stating that function history was 'window dressing', we have decided to implement it into FileWatcher. This was to ensure that we could properly reference which functions acted in what manner - helping us for testing, as well as for general use. Additionally, we created 'newfile' and 'delfile', which allowed us to delete files and create new files on the fly within the shell. This was considered essential, especially after filewatcher's creation (which allowed us to add better options for our argument).

Module 2

Error Handling:

Because of the disjointed nature of our system messager (as a result of the implementation of the fork process), there is cause for concern that failures found in the parent process would not affect the child process - when, in reality, this is the preferred approach. We want child processes to be preserved in the face of errors - this is to ensure the robustness of the product.

We also ensure that user inputs to the program are properly sanitised - as a result, we properly handle all errors that arise from bad input data. I

Exception Handling and Errno:

Instead of using exceptions and exception handlers, we have used contracts. This ensures a cohesive experience. Additionally, we did not use Errno - we decided that, instead of performing errno, we would use contracts to provide limitations on the user. Additionally, since no action the user can perform from our end would naturally yield an Errno reaction, we have forgone using it.

Robustness:

Ultimately, we came to the following conclusion for our implementation: if there is an external segmentation fault, then our system should keep monitoring files. Since users need to ensure that their programs are always monitored, we can ensure the robustness of the system.

Security:

We have decided to use contracts in lieu of exceptions in order to properly illustrate either bad inputs or bad commands. That way, we can properly handle improper inputs through 'requirements design', instead of using the exception hierarchy. However, if errors exist outside of this scope (most notably within C, where contracts have no reach), then the exception hierarchy within C is implemented.

Module 3

Sleep

In part three, we have swapped sleep(x) for nanosleep(x) - this is because of several reasons. First, we can ensure that the timing is more accurate by handling it by nanosecond count. This gives us a lot more leeway to ensure that our timing gives us the omst accurate response possible - without crashing on us. By doing so, we can increase precision 4+x that of sleep(x) (as we're using breaks of 250ms).

However, because of the nature of the second module (wherein a child process can simply sleep for an alloted period of time), the sleep(x) function will suffice. After looking at nanosleep, we decided that it would be easier for users to specify time delays in seconds - rather than milliseconds (i.e. the number of users that would benefit from 12000ms instead of 12 seconds outweigh the number of people that would use a shorter function with seconds). As a result, the second module retains its 'seconds' modifier.

Limitations

Filewatcher

Our filewatcher is built off of the premise that it can scan the whole directory and find the file. This limitation affects large file systems, as it may be significantly slower to process these than to process smaller file systems.

For filewatcher, only one action can be parsed at a time. For example, there can't be an LS and CD in one structure. For other functions, they work as designed.

Timing We have overhead on our processes and its count - when the system states it's checking immediately, there is a delay of 0.15s per item in the list. This is to ensure the cohesiveness of the product, but may not lend to accurate timing. While the 0.15s issue is relieved once the iterator is running, the overhead of reading through the file system is not accounted for, and may lead to minor timing problems.

We have used an iterator of 250ms per iteration for filewatcher. This is to ensure that the system doesn't take too many resources reading the files over and over, but also maximises the precision of the product.

Additional Testing

Unit Tests

Several unit tests were created to test he regex and the contracts. This helped us insure that our contract implementation was correct and had no gaps in in whatsoever.