Threads VS. Processes

Version 1.0

Eric Tsang, A00841554, 5D

|  |
| --- |
| Table of Contents [Table of Contents 1](#_Toc439886371)  [1 Finite State Machine 2](#_Toc439886372)  [1.1 Parent Process / Main Thread 2](#_Toc439886373)  [1.2 Child Processes / Worker Threads 3](#_Toc439886374)  [2 Data Flow Diagrams 4](#_Toc439886375)  [2.1 Threaded Version 4](#_Toc439886376)  [2.2 Multi-Process Version 6](#_Toc439886377)  [3 Pseudocode 7](#_Toc439886378)  [3.1 Parent Process 7](#_Toc439886379)  [3.1.1 main() 7](#_Toc439886380)  [3.1.2 handle\_task\_result() 7](#_Toc439886381)  [3.1.3 handle\_task() 8](#_Toc439886382)  [3.1.4 handle\_execution\_statistics() 8](#_Toc439886383)  [3.1.5 handle\_task\_completed() 8](#_Toc439886384)  [3.2 Child Process 8](#_Toc439886385)  [3.2.1 worker\_routine() 8](#_Toc439886386)  [3.3 Main Thread 9](#_Toc439886387)  [3.3.1 main() 9](#_Toc439886388)  [3.4 Worker Thread 9](#_Toc439886389)  [3.4.1 worker\_routine() 9](#_Toc439886390) |

# Finite State Machine

## Parent Process / Main Thread



|  |  |  |
| --- | --- | --- |
| State Name | Description | Events |
| Setup | The process or thread creates any IPC mechanisms, child processes (or worker threads), synchronization primitives and data structures needed for the *work state* of the program. | **Finish;** the application has finished all setup tasks, and moves on to the *work state*. |
| Work | The *main process (or thread)* posts the first work task into the *task queue* where it will be removed and processed by one of many *worker process* which may generate more *tasks* that will eventually be placed back into the *task queue* for processing. | **No more work;** the task pipe is empty, and all worker processes are idle. There is no more work to be done so the application moves on to the *cleanup state*. |
| Cleanup | The *main process (or thread)* prints its execution statistics and results to *standard out*, and proceeds to release any system resources it has acquired, then terminates. |  |

## Child Processes / Worker Threads



|  |  |  |
| --- | --- | --- |
| State Name | Description | Events |
| Get work | The *child process* or *worker thread* gets the next task from the task pool or blocks until one becomes available. | **Got work;** the *child process* or *worker thread* has received a task from the *task pool* and moves on to the *do work* state.  **No more work;** the *child process* or *worker thread* is notified that there will be no more tasks put into the task pool, so it moves on to the *dead* state, |
| Do work | The *child process* or *worker thread* performs the work specified by the task it has received. | **Finished work;** The *child process* or *worker thread* finishes the task it was executing and goes back to the *get work* state. |
| Dead | The *child process* or *worker thread* terminates. |  |

# Data Flow Diagrams

## Threaded Version



|  |  |
| --- | --- |
| Name | Description |
| Task Priority Queue | A data store that contains all the tasks that need to be processed by a worker thread. This data store orders tasks by priority so they are removed from the highest priority tasks, to the lowest.  Directory crawling tasks are the highest priority, followed by file hashing tasks, which are also prioritized by the size of the file to hash. larger files are given higher priority. Directory crawling tasks are given highest priority because we want to discover all the file hashing tasks asap. file hashing tasks on large files are given high priorities because they will take more time to process.  This prioritization scheme was chosen so when the application nears termination, ideally, the only tasks left in the task queue would be short tasks. So once the task queue is emptied, all worker threads would end promptly at around the same time. |
| Results | Data store of hash code of a file and the corresponding fully qualified path to the file. It is ordered in lexicographical order |
| Execution Statistics | Data store of worker thread execution statistics. |
| Main Thread | The main thread puts the very first job into the task priority queue, then waits until it, and all subsequently discovered tasks are finished before printing the execution statistics and hash codes out through standard out and terminating the program. |
| Worker Thread | The worker threads continuously get work to do from the task priority queue, and works on it. this produces execution statistics which they will put into the execution statistics data store, and may create more tasks, or results which they will then place in the appropriate data store. |

## Multi-Process Version



|  |  |
| --- | --- |
| Name | Description |
| Task Message Queue | Contains all the tasks. This is analogous to and does the same thing as the task priority queue in the threaded version of the program. |
| Feedback Pipe | Contains serialized task objects, execution statistics and hash code results written into it by worker threads. |
| Execution Statistics | Data store of worker thread execution statistics. |
| Results | Data store of task results (file hash codes). |
| Parent Process | The parent process places the first task into the task pipe, and then continuously reads from the feedback pipe, and handling the received information appropriately. once all tasks have finished being executed, it closes the task and feedback pipes, and prints the execution statistics and results to standard out. |
| Worker Process | Continuously receives tasks from the pipe queue, and executes them. When it gets EOF when trying to receive from the task pipe, it terminates itself. |

# Pseudocode

## Parent Process

### main()

Point of entry to the program.

|  |
| --- |
| 1. Create the pipes and data structures needed to send, receive and store tasks, task results, and execution statistics. |
| 1. Create the 5 child processes using the fan pattern. |
| 1. Close pipe descriptors such that we can only write tasks into the task pipe to the child processes, and read objects from the feedback pipe from the child processes. |
| 1. Write the initial task into the task pipe for the child processes to work on. |
| 1. While there are tasks being executed or to be executed by the application, read objects from the feedback pipe, and invoke the correct handler for the received object. |
| 1. Write the execution statistics to standard out, and the hash codes of files to standard out, and the user-specified file. |

### handle\_task\_result()

executed when a task result object is received from the feedback pipe.

|  |
| --- |
| 1. put the result object into the result data store |

### handle\_task()

executed when a task object is received from the feedback pipe.

|  |
| --- |
| 1. put the task object into the task message queue |
| 1. increment the task count |

### handle\_execution\_statistics()

executed when a execution statistics object is received from the feedback pipe.

|  |
| --- |
| 1. update the execution statistics from the information in the received execution statistics object. |

### handle\_task\_completed()

executed when a task completed object is received from the feedback pipe.

|  |
| --- |
| 1. decrement the task count |
| 1. if there are no more tasks left in the system (task count == 0), move on to the cleanup state. |

## Child Process

### worker\_routine()

main loop of the child process.

|  |
| --- |
| 1. Read and de-serialize data from the task pipe into a task object. If the pipe is closed, terminate the process. |
| 1. Do the work required by the task object. |
| 1. Get the new tasks, results, and execution statistics from the task object, serialize them, and write them into the feedback pipe. |
| 1. Write a task completed message to the feedback pipe. |
| 1. Repeat from step 1. |

## Main Thread

### main()

Point of entry to the program.

|  |
| --- |
| 1. Create the data structures needed to send, receive and store tasks, task results, and execution statistics. |
| 1. Create the 5 worker threads that run the worker\_routine() |
| 1. Write the initial task into the task priority queue for the worker threads to work on. |
| 1. Wait until the system runs out of jobs to process. |
| 1. Write the execution statistics to standard out, and the hash codes of files to standard out, and the user-specified file. |

## Worker Thread

### worker\_routine()

main loop of the worker thread.

|  |
| --- |
| 1. Get the next pending task from the task priority queue that needs processing. |
| 1. Do the work required by the task object. |
| 1. Get the new tasks, results, and execution statistics from the task object, and write them into the appropriate data store for each. |
| 1. Repeat from step 1. |