YSC3217 - Final Project

All your code should be contained in a zip file called posix_final_yourname.zip (e.g. `posix_final_perrault.zip` for me). The program must compile with the following flags: `-Wall –Werror`

This project is divided into multiple parts. **Each of these part should run on the Pi, as I will be grading using a Pi.**

Part 1: Scheduler (relevant code should be in scheduler.c and scheduler.h) [10 points]

In this part, you need to write a simple scheduler. The scheduler is a process that schedules execution for a specific number of processes.

At first, the scheduler creates a named pipe that can be used by other processes to be queued. When a process requests to be queued, it will be added to the queue. The scheduler will also issue a SIGSTOP signal to the newly added process to pause it.

Note: The size of the queue is a parameter submitted on the command line when running the scheduler (i.e. `-queue 5` will create a queue with exactly 5 slots)

As soon as the queue is full, the scheduler will allow each process to run for exactly one second. Please implement the following three strategies:

1. Round Robin with first process getting the first time slot
2. Round Robin with last process getting the first time slot
3. Random (each time slice, a random process [chosen from any of them including the one that was running previously] is allowed to run

Note: Strategies will be decided by an argument submitted one the command line (e.g. `-rrfifo, -rrlifo, -rrandom`)

Note: To pause a process, you should send it a SIGSTOP message, and use SIGCONT to resume execution.

At any point of time, a process may request to be removed from the queue, in that case, the scheduler will stop sending it signals, remove the process from its internal queue, and continue scheduling the n-1 processes remaining. When a process finishes, the scheduler should be able to store information about said process including:

- PID
- Number of time slices that the process ran

One every queued process is finished, the scheduler should print information about the number of processes that enrolled, the number of processes that deenrolled, and for each process that ran and successfully finished, its PID and number of time slices (such information about a process that deenrolled should be discarded).

The actual implementation can be done in any way you want, but you need to specifically implement the following functions (with correct names/arguments):

`int enroll(const char *pipe);` Called from a process which wants to be queued in the process queue from the scheduler. The argument is the access point of the named pipe on the file system. Should return a file descriptor.

`void deenroll(int fd);` Called from a process which wants to stop being scheduled. The request should remove the process from the list of enrolled processes. The argument is the file descriptor of the named pipe. Deenroll should also close the pipe.

Note: Be careful with writing on the pipe, make sure that only one process at a time can write on the named pipe!

Part 2: Binary trees (code should be in binary_tree.c and binary_tree.h) [10 points]

For this part, you will need to implement balanced binary trees.

A node in a binary tree has a number of children between 0 and 2 (included). A binary tree is balanced if for each node it holds that the number of inner nodes in the left subtree and the number of inner nodes in the right subtree differ by at most 1. A binary tree is balanced if for any two leaves the difference of the depth is at most 1.

Note: Each node of the tree should have a unique value.

Note: Children nodes on the left of a given node should all have a value lower than the current node. Nodes on the right should have a value higher than the current node.

Implement the following functions:

`tree * create_new_tree();` Allocates enough space for a tree.

`void balance_tree(tree *t);` Balances a binary tree.

`int add_node(tree * t, int val);` Add a new node storing val to the tree. Returns 0 if the insertion was done, -1 otherwise (e.g. if val is already a node).

`int edit_node(tree *t, int oldval, int newval);` Changes the value of a given node from oldval to newval. Returns 0 if the edit was succesfull and the tree was not changed (e.g. no nodes were moved), -1 if oldval could not be found, -2 if the tree had to be rebalanced.

`int remove_node(tree *t, int val);` Removes a node with value val. Returns 0 if successful, -1 if the value could not be found.

`int shortest_distance (tree *t, int node1, int node2);` Returns the minimum distance (in terms of numbers of nodes) between two nodes, -1 if any of the nodes does not exist.

`int find_depth(tree *t, int val);` Finds the depth of a given node. Root has a depth of 0, children to the root have a depth of 1, etc… Returns the depth (> 0) or -1 if the val could not be found.

`void print_nodes_at_level (tree *t, int level);` prints all the node found at level n. Note: level 0 is the root, level 1 should display the two children of the root, level 2 should display 4 nodes [grand-children of the node] etc…

`void print_tree(tree *t);` prints the tree in a user-friendly way.

`void delete_tree(tree *t);` Deletes a tree.

`tree * generate_random_tree(int size);` Generates a **balanced** tree with *size* nodes, each node has a random (unique) value. The tree must be balanced.

`tree * generate_from_array(int *array, int size);` Generates a balanced tree with *size* nodes. The values are stored in *array*.

Part 3: Processes at war (war.c and war.h) [15 points]

For this last part, you will implement a war-game using processes.

Your parent process should spawn X children. This parent process will be the one receiving information about the "state" of each child, and solving the combat phase.

Children

Each child should create Y soldier threads and Y farmer threads. Farmers are threads that should produce resources. The general behavior of a farmer is as follows:

- Sleep for a random duration between 400 and 2200 ms
- Produce a resource stored in a buffer. Producing one resource means incrementing a variable by one.
- **Note:** There should only be a maximum of 6 farmers active at a time.

Soldiers behave as follows:

- Once per second, the main thread of the child will wake up the soldiers. Each resource will be consumed by the soldiers and be translated as one attack point. Consuming a resource means decrementing the counter of resources by one.

The main thread of each child works as follows:

- Create the Y soldiers and Y farmers. **There should only be a maximum of 6 (Y > 6) farmers active at a time. The active farmers are chosen randomly each iteration.**
- **Resources are shared between soldiers and farmers. (one soldier consumes the resource created by one specific farmer. Access to that data must be synchronized).**
- Once each soldier has communicated its attack points, the attack points are aggregated. The main thread then chooses one of the available opponents to attack (another child process). The number of attack points + opponent ID is communicated to the parent process.

Parent

After creating X children, the parent will simply wait for each individual child to send information about which other child they want to attack + attack points.

The damage taken by each individual child is simply the number of attack points received from the other children. If two processes were to attack each other, the damage point would be the difference of attack points from both processes. The process with the lowest attack points would be taken the damage, while the winner would not take any.

The parent should log every attack in a log file containing the current date/time, the attacker, the damages inflicted and the receiver of the damage. The log file should contain the number of active soldiers per child process after every round of battle.

The log file should also contain entries when a process is out of soldiers (it then dies). **The parent should wait for each child to finish, and print a statement in the terminal as well (e.g. "Child #XXX has died").**

Damage and End

Damages taken are split between the soldiers. Each soldier can take up to three damages. Every time a soldier dies, its associated farmer can start working for another soldier. **Note: in that case, the new farmer need to also get exclusive access to the resources when creating resources.**

As soon as a process is out of soldiers, it dies and finishes. Its main thread should make sure that every farmer/soldier threads (if relevant) are finished and print a statement **(e.g. "Farmer thread #XXX finished.").**

Other processes are not allowed to attack a dead process.

The game stops when there is only one child process still running. The log file will contain a final line with date/time and the PID of the winning child (e.g. "08:54 >> Child #XXX won the game!").

Command Line Arguments

The main program will take two arguments on the command line: `-children X` **and** `-fighters Y` **(note that there are as many farmers as soldiers per child)**

Synchronization Mechanisms and Interprocess Communication

You are free to use any synchronization or interprocess communication mechanisms. For interprocess communication, please make sure that only one individual process is allowed to access said mechanism at any given time.

Part 4: Short report (report.pdf) [5 points]

In this report, you will briefly explain which mechanisms you used for Part 3, and justify your choices. Note that there are multiple valid solutions. If your solution could have some potential issue, please briefly explain what the issue is and how you solved it/this could be solved.