COMP 2004
Instructor: Dr Vinicius Prado da Fonseca
Assignment 2 - Fork

1. (50%) Assume one parent process has 3 child processes that are forked, then joined using 3 different join constructs.
    a. The **join_none** construct makes the parent not wait for any child processes to finish before resuming the parent process, i.e. it runs concurrently with the child processes.
    b. The **join_all** construct makes the parent wait for all child processes to finish before resuming the parent process.
    c. The **join_any** construct makes the parent wait for at least one child process to finish before resuming the parent process.

    The solution of the first mode is provided in the assignment. The assignment contains 2 files:

    - **task.c** is a simple console application that sleeps for 10 seconds, marking the start/end of the process. Compile this file using:

      `gcc task.c -o task`

    - The **join_none.c** file provided is the solution of the first mode as explained above. Notice that all 4 tasks overlap concurrently. Compiling: `gcc join_none.c -o join_none` and running: `./join_none` the output looks like this:

      ```
      I'll wait for no one!
      parent 0 start ...
      task 1 start ...
      task 2 start ...
      task 3 start ...
      parent 0 end
      task 1 end
      task 2 end
      task 3 end
      ```

    Note that the parent starts right away.

    Write two other C console programs (**join_all.c** and **join_any.c**) that will fork and join the processes similar to descriptions b and c above. The output should look like this:

    - **join_all.c**

      ```
      I'll wait for everybody!
      task 1 start ...
      task 2 start ...
      task 3 start ...
      ```

```
task 1 end
task 2 end
task 3 end
parent 0 start ...
parent 0 end
```

Note that the parent starts after everybody finishes.

- **join_any.c**

```
I'll wait for one!
task 1 start ...
task 2 start ...
task 3 start ...
task 1 end
parent 0 start ...
task 2 end
task 3 end
parent 0 end
```

Note that the parent waits for at least one child and then starts.

2. (50%) An operating system's pid manager is responsible for managing process identifiers. When a process is first created, it is assigned a unique pid by the pid manager. The pid is returned to the pid manager when the process completes execution, and the manager may later reassign this pid. Process identifiers are discussed more fully in Section 3.3.1.

   What is most important here is to recognize that process identifiers must be unique; no two active processes can have the same pid.

   Use the following constants to identify the range of possible pid values (for testing these values may be changed, your code must be able to deal with the changes):

   ```
   #define MIN_PID 300
   #define MAX_PID 5000
   ```

   You must use a linked list. Read the linked lists pdf and review the code in the course material.

   Implement the following API for obtaining and releasing a pid:

   `void init_list(void)` - Creates and initializes a data structure for representing pids; (Hint: review what is meant by an empty linked list. The solution here is trivial, don't overthink)

`int allocate_pid(void)` - Allocates and returns a pid; returns −1 if unable to allocate a pid (all pids are in use)

`void release_pid(int pid)` - Releases a pid

In the main function of your program create test routines that allocate and release pids to test your API implementation. This API/structure may be used in later assignments. You may use the provided sample “a2q2_sample.c”