Server

1 Server

This exercise is about parallelizing a server that receives a given number of requests of different type, processes each of them and stores the results on the appropriate stack. The corresponding sequential code is as follows:

```
for(;;){

  req = receive();

  printf("Received request %d\n",req.id);
  if(req.type != -1) {

    /* process request and push result on stack */
    printf("Processing request %d\n",req.id);
    stacks[req.type].results[++stacks[req.type].head] = process(&req);

  } else {
    break;
  }

  printf("Finished \n");

}
```

This code relies on two data structures. The first is `Request`:

```
typedef struct requeststruct{
  int    type;
  int    id;
  double data;
} Request;
```

and contains three fields indicating the type (`type`), the request identifier (`id`) and some data (`data`) which is used when the request is processed. The second is Stack:

```
typedef struct stackstruct{
  int head;
  Result *results;
} Stack;
```

which implements a stack data structure that stores data of type `Result`. The `head` field points to the top of the stack, i.e., the latest result stored therein; therefore, when something is stored on top of the stack, this value must be incremented.

The server enters a loop where, at each iteration, it receives a request `req` of type `req.type`. It processes it making a call to the `process` routine, stores the result on the stack `stacks[req.type]` of the corresponding type. The execution is terminated when a request of type -1 is received.

2 Package content

In the `server` directory you will find the following files:

- `main.c`: this file contains the main program that first executes the sequential code presented above. **Only this file has to be modified for this exercise**.
- `aux.c, aux.h`: these two files contain auxiliary routines and **must not be modified**.

The code can be compiled with the `make` command: just type `make` inside the `server` directory; this will generate a `main` program that can be run like this:

```
$ ./main
```

3 Assignment

- Parallelize the code **using the OpenMP task directive**. In the paralle code there must be a server process that receives all the requests and makes tasks for them. All the threads (including the server) will execute the created tasks.