Assignment 1: Interprocess Communication

2INC0 - Operating systems

From

Full Name	Student ID
Daniel Tyukov	1819283
Nitin Singhal	1725963
Ben Lentschig	1824805

Eindhoven, December 4, 2024

CLIENT: The client is responsible for generating job requests and sending them to the router-dealer using the message queue provided at startup. It begins by opening the request queue (`Req_queue_T21`) with write-only permissions. The name of this queue is passed as an argument when the client process is created by the router-dealer. The client then enters a loop where it retrieves job requests by invoking the `getNextRequest()` function, which returns the job ID, data, and the requested service ID. For each request, the client formats a message containing this information and pushes it to the request queue.
If the request queue is full or any issue occurs during message sending, the client logs the error to `stderr` and retries on the next iteration. Once all requests are processed (as indicated by `getNextRequest()` returning `NO_REQ`), the client releases its resources, closes the message queue, and terminates. The pseudo-code summarizing the client logic is as follows:

```
1 setup();
2 open request queue (write-only);
3 while (true){
4     result = getNextRequest();
5     if (result == NO_REQ){
6         break; // exit loop when no more requests
7     }
8     prepare request message;
9     if (mq_send(request queue) == -1){
10         log error to stderr;
11         continue; // retry in the next iteration
12     }
13 }
14 close request queue;
15 cleanup resources;
```

ROUTER_DEALER: The router opens queues non-blockingly then enters a loop where every iteration it attempts to receive from client in a circular buffer, attempts to send to workers then attempts to print their response. Whenever a task is failed, the router tries again in the next loop. A buffer is used to accommodate for the client which is faster than workers. Workers are created during setup and continue working until the router signals them to terminate with a job with id -1. The router exits its loop when it detects a terminated client and that all jobs received were processed. The router maintains statistics to track buffer pointers/count, number of jobs received and processed. The following pseudo code summarizes the logic.

```
setup();
while (1){
    if(space in buffer){
        if (mq_receive(client queue)) update_statistics;
    }

    if(items in buffer){
        if (service_id == 1){
            process_job;
            if(mq_send(worker1 queue) == 0) update_statistics;
        }
        // Send to worker_s2 queue ...
    }
    if (mq_receive(response) == 0){
        printf(result);
        update_statistics;
    }

    if(client is running) waitpid(clientID, &client_status, WNOHANG);

    else if (num_processed==num_recieved) break;
}
send_kill_signals_to_workers();
cleanup_all_resources();
```

WORKER: The worker process was designed to act with little agency, as it completes the requested work on a simple call-and-execute basis. This means the worker does not perform any sanity checks on sizes, inputs, outputs, or validity of outputs. The worker only validates everything related to message queues (opening, closing, receiving, and sending). The worker sets up message queues and then continuously runs in a loop checking for work to be done under certain conditions until the router dealer requests termination. Pseudo-code of this implementation is below:

```
setup();
while(1){
    if(previous job sent) // condition passed on first iteration
    {
        try_receive_new_job();
    }
    elif(new job not received){
        retry_to_recieve;
        continue;  // skips rest of code to retry
    }
    if(termination not requested){
        send_response();
    }else{
        break; //breaks out of the loop
    }
}
close_queues();
```

DEADLOCK ANALYSIS: Since all queues are in non-blocking mode the processes will never wait for anything and never enter a circular deadlock (all processes waiting for resources held by other processes). The data structure supplying getNextRequest() in the client while loop condition is finite, it will eventually return NO_REQ so the client always eventually exit. To prove absence of livelock, we assume a livelock situation and derive contradictions. The variables "q_client", "q_work", "buff" etc track fullness of data structures. During a livelock, these variables must be invariant regardless of which process executes which statements (else progress is made). Thus, we can make a wait-for graph where each process loop executes atomically. Arrows of the graph denote conditions under which the pointed to process will busy-wait. Multiple arrows pointing to an object compose as AND, ex - Router busy waits if (E1 AND E2). The arrows on the graph depicted are **all possible busy-waiting conditions**.

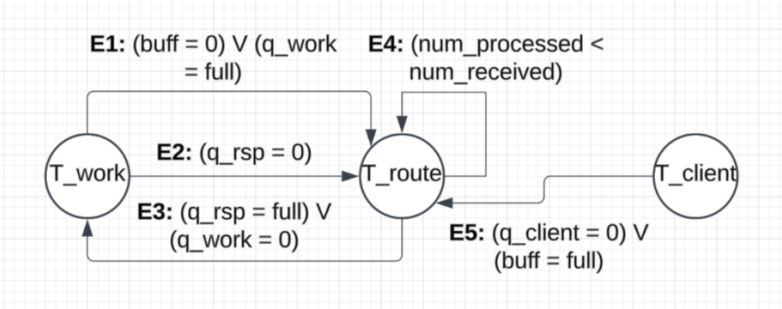

Figure 0.1: Wait-for diagram

E1 is derived from router code line 7 and 10 (jobs can only be sent if there is any to send in the buffer, and if the queue can take it). Thus with E1, the router can't make progress by forwarding jobs. E2 is derived from line 14 (response queue needs items to be popped). E3 is derived from worker code line 5 and 12 (the worker can only progress if it can pop from worker queue or push to response queue). E4 is the exit condition of the router loop (line 21) (only if the client is not running). E5 is derived from router line 4 and 5.

Since the response queue cannot be both full and empty, E2 and E3 imply that the worker queue is empty. The empty worker queue, and E1 then imply that buff=0. buff=0 and E5 implies that the client queue is empty. Thus, we reach the conclusion that all queues and the buffer must be empty, and num_processed < num_received. This is the only consistent livelock state.

This state can only be reached when a job is dropped by the router or worker. The router can only drop jobs by overwriting one in the buffer. The router buffer is protected against overwriting by line 3, which enforces a capacity limit. The worker is prevented from dropping a job with line 3, which requires the previous job to be successfully sent before overwriting it. Thus, the code is deadlock free and does not busy wait.