Assignment 1: Interprocess Communication

2INC0 - Operating systems

From

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

Eindhoven, December 3, 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;
```

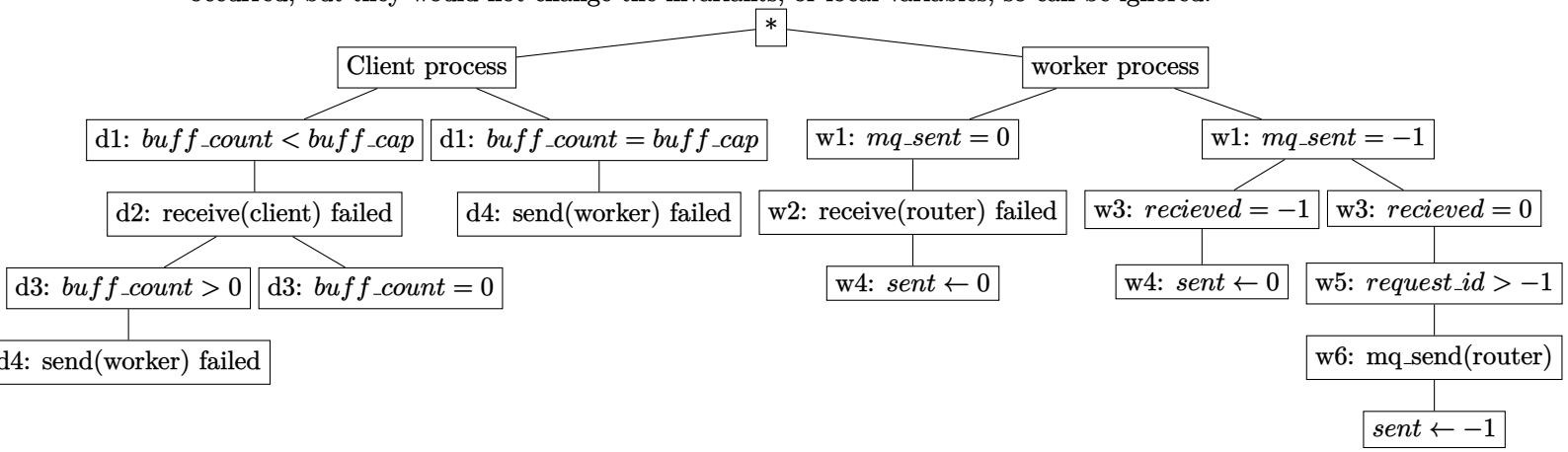
ROUTER_DEALER: The router opens queues non-blockingly then enters a loop where every iteration it attempts to receive from client in a circular buffer, attempts to send to workers then attempts to print their response. Whenever a task is failed, the router tries again in the next loop. A buffer is used 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){
        do_work();
    }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 that a livelock (no progress despite looping) is impossible, We assume an **existing** live lock condition and derive a contradiction. In each process, traces in the **while loop** are made into a binary tree of pseudo-code (some variable names are changed). The queue and buffer fullness have are assumed **invariant** (if they changed, it would not be live lock) and the processes won't exit. Between edges in the tree, instructions from other processes could have occurred, but they would not change the invariants, or local variables, so can be ignored.



For each router branch, $receive(response)$ must also fail (so empty response queue) and $num\_processed < num\_received$ so there is no loop exit. d2 implies empty client queue, d3 implies full while w2 implies empty worker queues. w6 implies a full response queue. In the workers, w4 in the second branch forces the process to return to the first branch. Thus it inherits the constraints on invariants of the first. The variables track fullness of data structures. Constraint on invariants are shown as logical statements.

Router Branch 1 $((0 < buff < full) \wedge (q\_req = 0) \wedge (q\_worker = full) \wedge (q\_rsp = 0))$

Router Branch 2 $((buff = 0) \wedge (q\_req = 0) \wedge (q\_rsp = 0))$

Router Branch 3 $((buff = full) \wedge (q\_worker = full) \wedge (q\_rsp = 0))$

Worker Branch 1,2 $((q\_work = 0)$

Worker Branch 3 $(q\_rsp = full)$

Branch 3 of the worker which needs $(q\_rsp = full)$ is incompatible with all of the router branches which need $(q\_rsp = 0)$ so is eliminated. Branches 1 and 2 of the worker, needing $(q\_work = 0)$ are compatible only with branch 2 of the router. Thus livelock only occurs when all queues/buffers are empty and $(num\_processed < num\_recieved)$. This only happens 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 d1, which enforces a capacity limit. The worker is prevented from dropping a job with w1. Only on branch 1 can the previous request be overwritten. It can only enter branch 1 if it successfully sent the previous job (mq_send is set 0 on branch 3), or if it failed to read the previous job (branch 2).