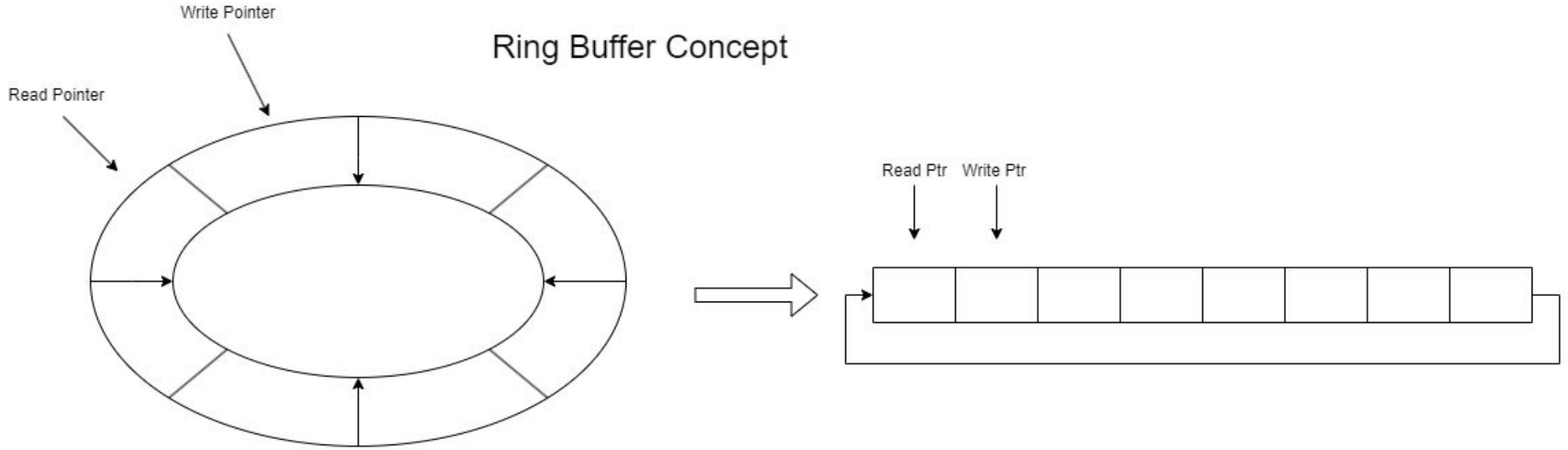
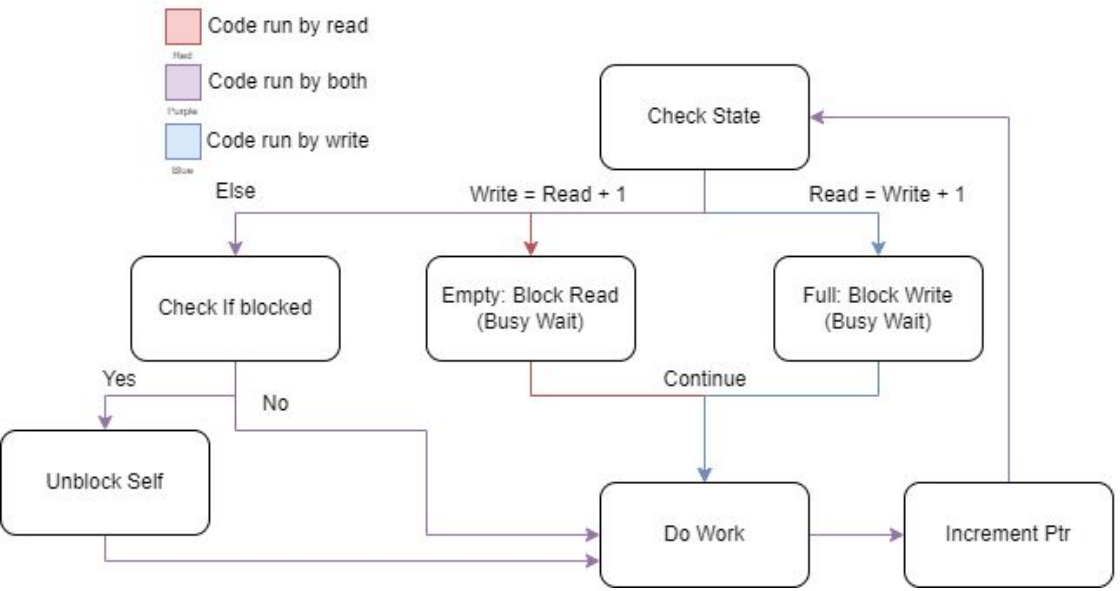
Άσκηση 1

Γράψιμο/Ανάγνωση στο/από το pipe γίνεται με τη χρήση read/write pointer.

- Pipe_write γράφει 1 byte στην τρέχουσα θέση τουwrite pointer και έπειτα προχωράει μια θέση.
- Pipe_read διαβάζει 1 byte από την επόμενη θέση και προχωράει κατά μια θέση.
- Πάντα 2 κενές θέσεις στο buffer.
- Το νήμα που τρέχει pipe_write μπλοκάρει αν στην επόμενη θέση από τον write pointer βρίσκεται ο read.
  - Γεμάτος buffer. Περιμένει κάποιο άλλο νήμα να διάβασει από pipe
- Το νήμα που τρέχει pipe_read
- Pipe_open δεσμεύει μνήμη για δομές απαραίτητες για την υλοποίηση και αρχικοποιεί
- Pipe_writeDone αλλάζει σε 0 το flag open_to_write





Unwanted State: Read ahead of Write
Potential Bug: re-writes buffer if write doesnt catch up

Σε μία "χαζή" υλοποίηση θα αφήναμε τους 2 pointer
να ακουμπάνε και θα είχαμε κάποιο flag Full-Empty για να block-άρουμε
τις read και write την οποία θα πρέπει να θέτουν πότε η read και πότε
η write. Ετσι 2 thread ακουμπάνε την ίδια μεταβλητή και έχουμε τέτοια bugs
Συνεπώς χρησημοποιήσαμε την παραπανω υλοποίηση

Κρίσιμος κώδικας

pipe_write()

….

/* Pipe full */

WHILE write index + 1 same as read index {NOP}

Write character c in to buffer at current write index

…..

pipe_read()

….

/* Pipe emty open for writting */

WHILE (read index + 1 is the same as write index and pipe p is open for writing) { NOP }

/* Pipe emty closed for writting*/

IF pipe is empty and closed for writing {

Remove pipe p

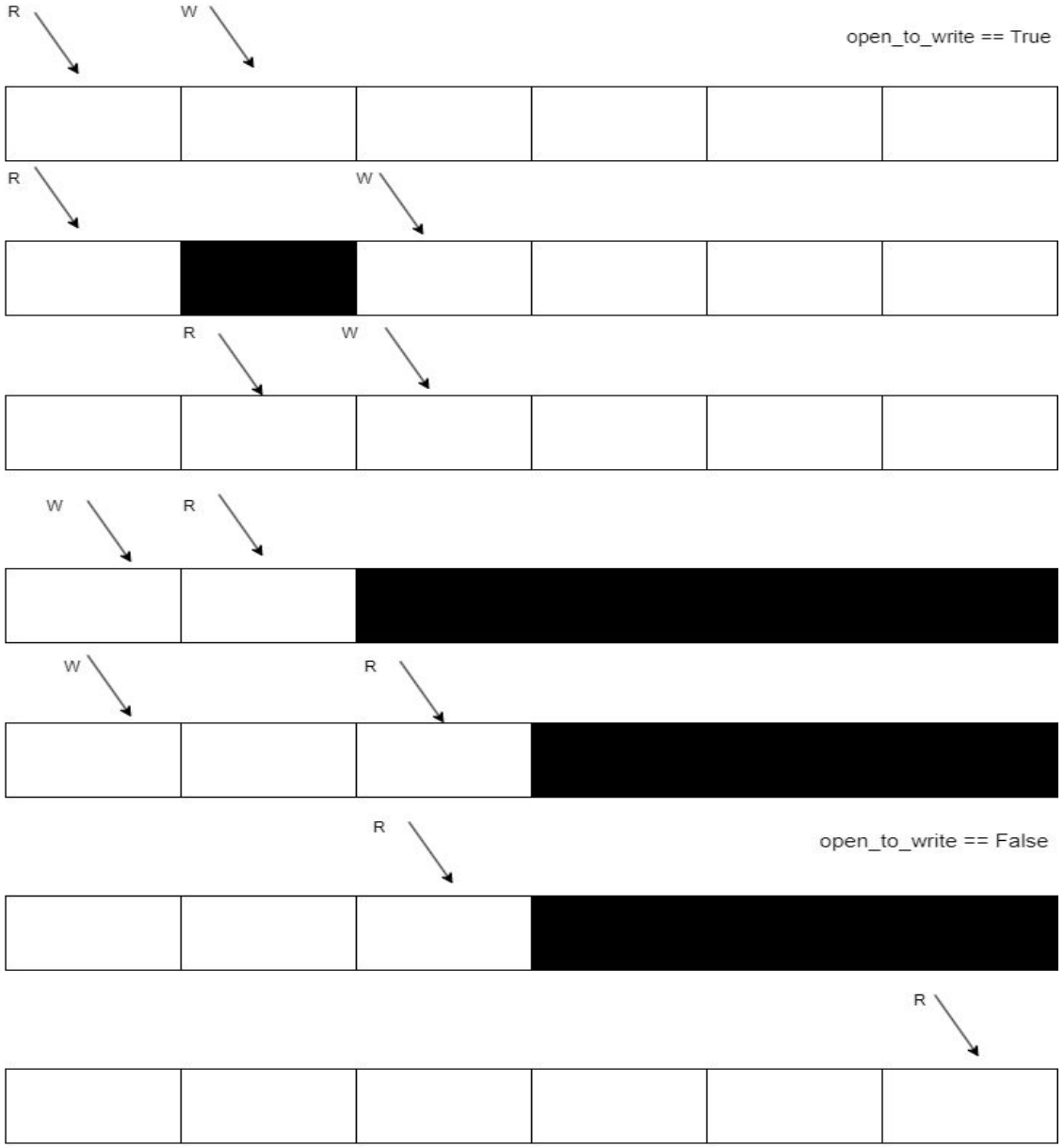
RETURN 0

}

Read character from read index + 1 of buffer and assign it to c

….

Παράδειγμα εκτέλεσης



Άσκηση 2

- Ο συγχρονισμός ανάμεσα στο main νήμα και στα νήματα εργάτες γίνεται μέσω εντολών που στέλνονται από τη main και ορισμού κατάστασης από τα νήματα εργάτες.
- Στην ουσία κάθε νήμα εργάτης έχει
  - ένα πεδίο state που ορίζει την κατάσταση του (AVAILABLE, WORKING ή TERMINATING) .
  - και ένα πεδίο command που δείχνει τη πιο πρόσφατη εντολή που δέχτηκε από το main (WAIT, PROCESS_VALUE, TERMINATE).

Κρίσιμος κώδικας

Main thread

```
…..

WHILE end of input is not reached {

    Read one integer from users input

    DO {

        Available_worker_id <- check_available_worker(workers, N)

    } WHILE no workers are available

    Assign input number to available worker's value_to_process

    Write to available worker's command, PROCESS_VALUE flag

    WHILE (worker i is still available but a new value to be processed has
    been assigned to it) { NOP }

    }

FOR every worker i

        Set worker's i command to TERMINATE

WHILE at least one worker hasn't finished yet { NOP }

…..
```

Worker thread

```
……

WHILE true {

        Set current worker thread state as AVAILABLE

        WHILE current thread's commad is WAIT { NOP }

        IF current thread's command is TERMINATE {
            Set current thread's state to TERMINATING

            BREAK

            }

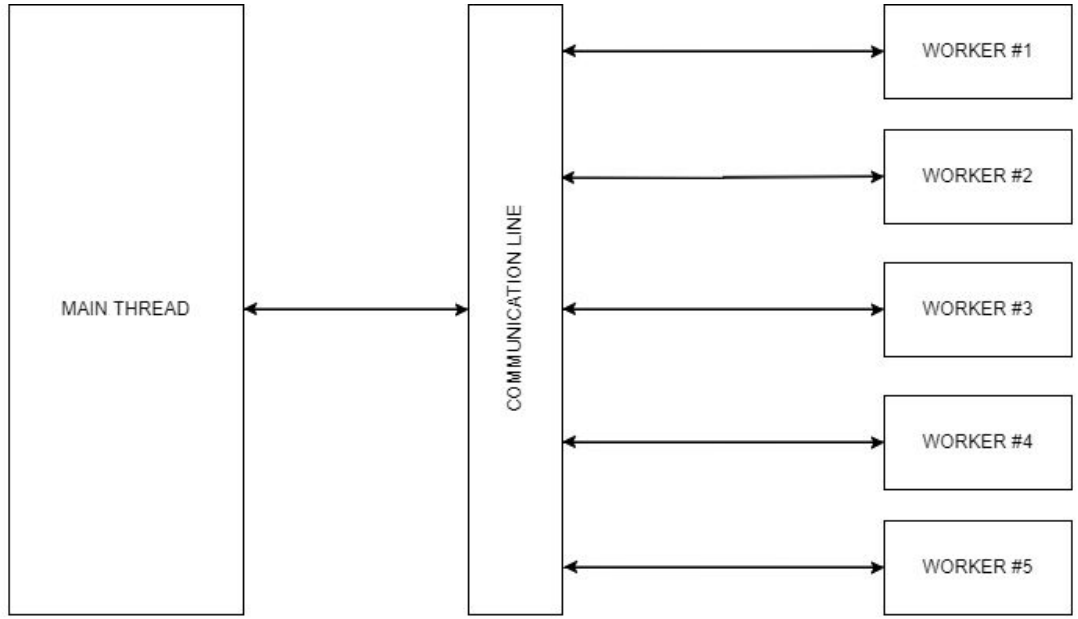
            Set current thread's command to WAIT

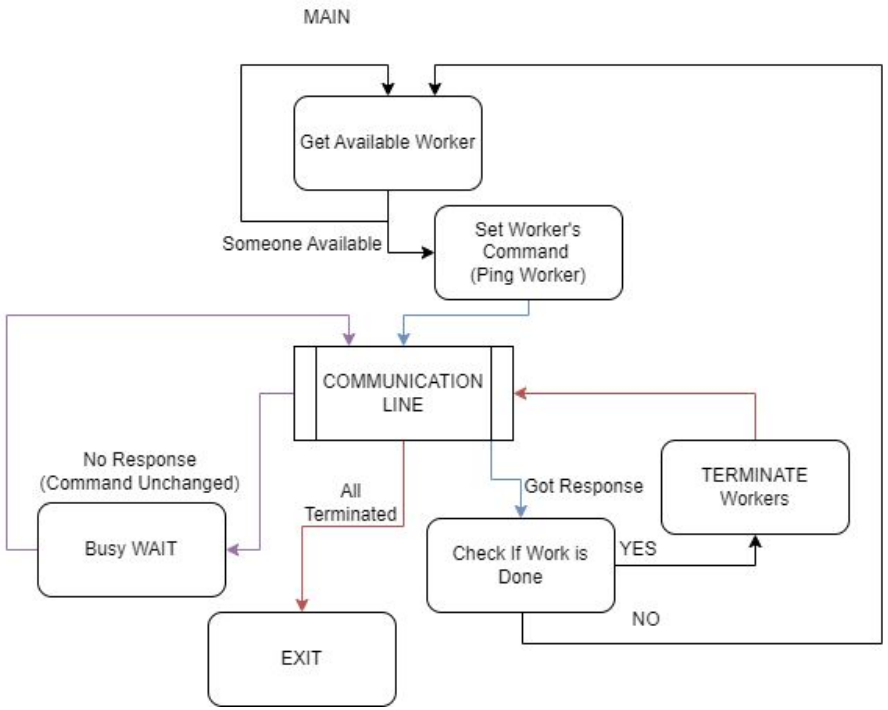
            Set current thread's state to WORKING

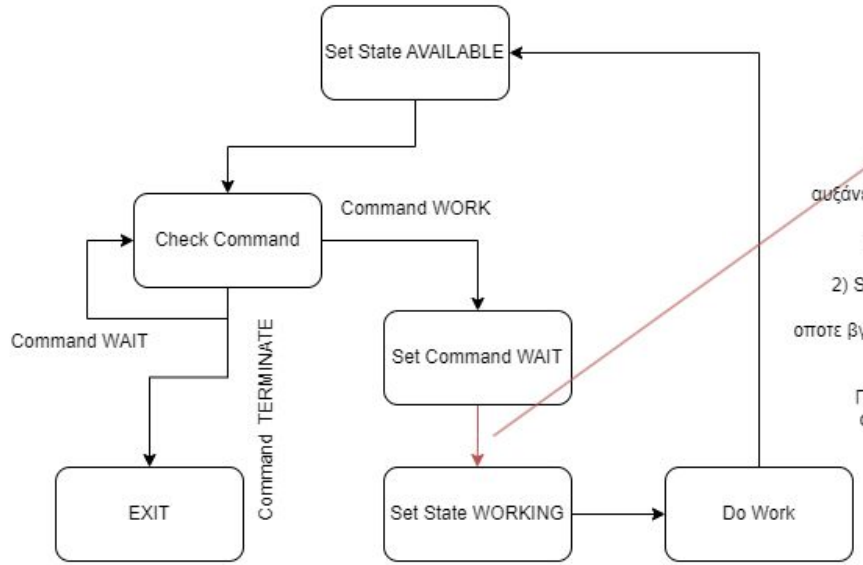
            Calculate if value_to_process of current thread is prime
            or not

            Print result

}
```







BUG
POTENTIAL FIXES (HAD no time to deploy)

1) Χρήση ενός counter μαζί με το state του worker το οποίο αυξάνεται από τον worker όταν τελειώνει ένα job. Έτσι η main ξερει όταν ο worker που του ανέθεσε ένα job ξεκίνησε να δουλεύει ή τελείωσε το job (σε περίπτωση που η main δεν παρει τον επεξεργαστή όσο ο worker δουλεύει)
2) Swap WAIT and WORKING command-state assignments το οποίο δημιουργεί το πρόβλημα του overwrite TERMINATE command τησ main οποτε βγαζουμε το command TERMINATE απο το struct και το κανουμε global BOOLEAN

Παρατηρούμε οτι δεν υπαρχει πρόβλημα αναμεσα στο WAIT και WORK όταν ο worker θετει τον εαυτό του WAIT αφού η δουλειά είναι πλέον αναπόφευκτη

Άσκηση 3

Ένα νήμα περιμένει τα νήματα παιδία να ταξινομήσουν το κομμάτι του αρχείου που τους αντιστοιχεί. Ο συγχρονισμός γίνεται με την μεταβλητή running που περνάει σαν όρισμα στα νήματα. Το flag running γίνεται 1 όταν δημιουργείται ένα νήμα παιδί και τρέχει κώδικα με τον οποίο ταξινομεί το αρχείο. Αλλάζει σε 0 όταν τελειώσει την δουλεία του.

Κρίσιμος κώδικας

```
FUNCTION void *run_thread(void *arg) {
        Get info for current thread from parent thread

        IF size of partition is equal or smaller 64 ints {

                Allocate memory for a interger's array with size ints

                Read ints from start to end from file and store them in the buffer

                Sort array

                Write the sorted array back to the file

                Free allocated memory used for the buffer

                Set running flag to 0

                RETURN NULL

    }
```

Κρίσιμος κώδικας συνέχεια

Calculate mid index of current patition

Dynamically allocate memory for info of child threads

Initialize info for left child thread, start <- start, end <- mid-1

Initialize info for right child thread, start <- mid, end <- end

create threads

WHILE left and right child are running { NOP }

Free allocated memory for child thread info

}