Άσκηση 1

Αρχικά, καλούμε την pipe_init για αρικοποίση του αγωγού μεγέθους size που δίνεται ως όρισμα και του πεδίου valid σε 1 έτσι ώστε όλες οι θέσεις να είναι έτοιμες για γράψιμο. Άν το πεδίο valid είναι 2 τότε η θέση είναι διαθέσιμη για διάβασμα. Δημιουργούμε δύο threads, ένα για γράψιμο κι ένα για διάβασμα. Μέσω αυτών, καλούνται οι συναρτήσεις pipe_read/write/close. Η pipe_read μπλοκάρει όταν ο αγωγός είναι άδειός(valid==1) ή μέχρι να κλειθεί η pipe_close για να επιστρέψει 0 και η pipe_write όταν ο αγωγός έιναι γεμάτος(valid==2). Ανεπιθύμητες συνθήκες ανταγωνισμού προκύπτουν όταν read και write προσπαθούν να προσπελάσουν την ίδια θέση του αγωγού. Το πρόβλημα αυτό λύνεται με το πεδίο valid που επιτρέπει μόνο σε μια από τις δύο διεργασίες να βρίσκεται στο κρίσιμο τμήμα κάθε στιγμή. Όταν η write_thread λάβει EOF καλεί την pipe_close που αλλάζει την τιμή της global μεταβλητής close_val από 0 σε 1, για να ενημερώσει την pipe_read να επιστρέψει 0 στην read_thread. Έπειτα, η read_thread αλλάζει πάλι το close_val σε 0 και καλέι την pipe_read για να διαβάσει τα τελευταία δεδομένα, εφόσον αυτά υπάρχουν. Έτσι, αποφεύγονται προβλήματα ανταγωνισμού μεταξύ των pipe_close και των άλλων συναρτήσεων.

pipe_write

```
while(data[i].valid != 1){
      yield()   //pipe_write is waiting
}

process data

data[i].valid = 2  //pos i is ready for reading
```

pipe_read

```
while(data[i].valid != 2){
      yield()  //pipe_read is waiting
}

process data

data[i].valid = 1    //pos i is ready for writing
```

Το write_thread μόλις λάβει EOF καλεί την pipe_close και τερματίζει. Το read_thread τερματίζει όταν διαβάσει και τα τελευταία δεδομένα αλλάζοντας την μεταβλητή της main (term) από 0 σε 1 για να την ενημερώσει ότι μπορεί να τερματίσει. Η μεταβλητή αυτή δίνεται ως όρισμα μέσω της pthread_create για το read_thread.

Άσκηση 2

Αρχικά, το πρόγραμμα παίρνει ως όρισμα τον αριθμό των threads που θα δημιουργηθούν. Το κάθε διαθέσιμο thread παίρνει έναν αριθμό από την main και εξετάζει αν είναι πρώτος μέσω της συνάρτησης primetest (return value=1). Όταν ένα thread υπολογίσει το αποτέλεσμα είναι διαθέσιμο να πάρει και άλλον αριθμό,εφόσον υπάρχει. Το πρόγραμμα σταματά να παίρνει είσοδο μόλις δεχθεί αριθμό <= 1 καθώς δεν θεωρείται έγκυρος. Για την υλοποίηση μας, χρησιμοποιήσαμε ένα struct για τον κάθε worker με τρία πεδία, thread_id, value και status. Όσον αφορά στο status, μπορεί να πάρει ακέραιες τιμές από -2 έως 1, 0 για να δηλώσει ότι είναι διαθέσιμος και 1 για να δηλώσει ότι δουλεύει. Όταν η main θέλει να τους ειδοποιήσει να τερματίσουν (εφόσον έχει τελειώσει το input), περιμένει όλα τα status να γίνουν 0, έτσι ώστε να τελειώσουν όλοι οι αριθμοί προς έλεγχο, και τα αλλάζει σε -1. Έπειτα, περιμένει από αυτούς να το αλλάξουν σε -2 για να βεβαιωθεί ότι τερματίζουν και τερματίζει κι η ίδια.

Main thread:

```
create the workers with status=0
while(input <= 1){
        wait for an available thread //status=0
        //assign job to the available thread
        value=input
        status=1 //process the job
}
wait for every thread to be available
status=-1 //notify workers to terminate
wait for all workers to terminate
```

Worker thread:

```
while(1){
        if(status=1){
                primetest //calling primetest
                status=0 //notify that I am available
        }
        if(status=-1){
                break //notified to terminate
        }
        yield() //waiting for notification
}
status=-2 //notify main that I will terminate
```

Όσον αφορά στην απόδοση του προγράμματος σε συνάρτηση του αριθμού των threads, έπειτα από μετρήσεις, παρατηρήσαμε ότι ο χρόνος εκτέλεσης, όσο αυξάνονται τα threads, μειώνεται μέχρι να φτάσει σε ένα διάστημα τιμών για το πλήθος των threads στο οποίο οι αντίστοιχες αποδόσεις έχουν πολύ μικρή απόκλιση μεταξύ τους (κάτι που μπορεί να οφείλεται στους αριθμούς εισόδου και στο αν σε κάποιες περιπτώσεις τα threads τερματίζουν πολύ κοντά το ένα με το άλλο). Αμέσως μετά, ο χρόνος εκτέλεσης συνεχώς αυξάνεται επειδή γίνονται πολλές εναλλαγές και ενεργές αναμονές που επιβαρύνουν το σύστημα.

Άσκηση 3

Το πρόγραμμα παίρνει ως είσοδο μία ακολουθία ακεραίων που αποθηκεύει σε έναν καθολικό πίνακα. Η main δημιουργεί το thread στο οποίο αναθέτει ολόκληρο τον πίνακα προς ταξινόμηση. Το κάθε thread (που επεξεργάζεται ένα τμήμα του πίνακα) με την σειρά του, εφόσον τοποθετήσει το στοιχείο διαχωρισμού στην τελική του θέση, και με βάση αυτήν, διαχωρίζει το υποτμήμα του πίνακα που ελέγχει σε δύο μέρη και αναθέτει το καθένα σε ξεχωριστό thread, που δημιουργεί και καταστρέφει δυναμικά. Αν ένα υποτμήμα είναι μικρότερο του 2, το νήμα τερματίζει χωρις να κάνει τίποτα. Ένα νήμα περιμένει τα παιδιά του να τερματίσουν (curr_left→ready == 1, curr_right→ready == 1) για να μπορέσει να τερματίσει και το ίδιο και να δώσει την επιμέρους λύση στον γονιό (curr→ready == 1).

Main:

```
while(1){
        take input
        if(EOF || numbers == max){
                adjust right pos
                break;
        }
}

make first thread
Wait for the first thread to terminate
print
```

Worker thread(quicksort):

```
pivot = array[curr→right]
while(1){
        while(array[++i]<pivot){
        }
        while(pivot<array[--j]){
                if(j == curr→left){
                        break
                }
        }
        if(i>=j){
                break
        }
        swap(i, j)
}
swap(i, curr→right)
make right child if necessary
make left child if necessary

wait for childs to terminate
free childs
```