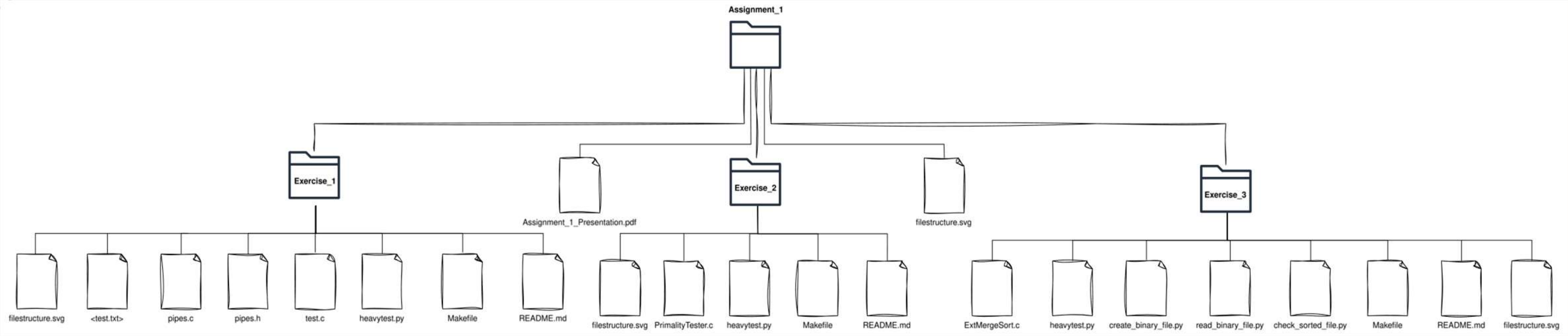
CONCURRENT PROGRAMMING ASSIGNMENT 1 – ACTIVE WAITING

Dimitris Voitsidis 03480
Iordana Gaisidou 03570
Stavros Stathoudakis 03491

PROJECT FILE STRUCTURE



1.1. FIFO pipes

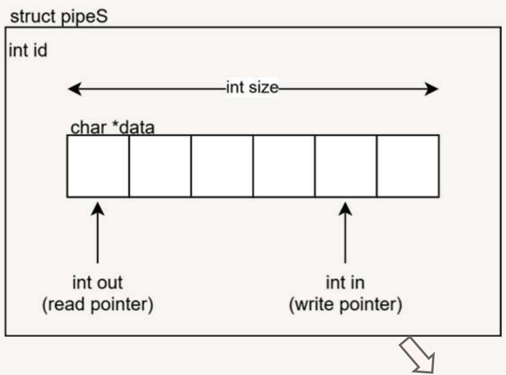
1.2. Primality Tester

1.3. External Mergesort

1.1. FIFO PIPES

Data structures

```
struct pipe {
  int id // id of the pipe
  int in // input index of pipe data
  int out // output index of pipe data
  writeE write // access for write
  int size // size of data array
  char* data // data array
}
```



```
struct threadArgS {
    int pipein // Pipe to write
    int pipeout // Pipe to read
    char* filename // Initial file
    bool  returned // Done flag
}
```

write flag
OPEN or CLOSED based
on pipes write state

Global Variables

```
pipeS** pipe_arr;  // Array of pointers to pipes
int pipe_arr_size;  // Size of the array
int pipes_ctr;        // Counter for ids of pipes
```

Functions

1) Pipe Open

```
function pipe_open(size):
    create new pipeS object p
    allocate memory for p and p->data
    initialize p with id, in, out, write, size
    add p to pipe_arr
    increment pipe_arr_size and pipes_ctr
    return p->id
```

Called by Main twice in test.c
Creates pipe1, pipe2

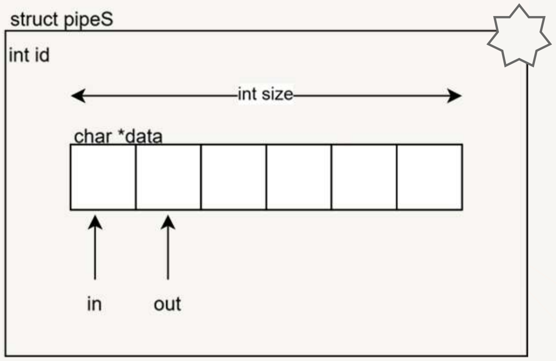
1.1. FIFO PIPES

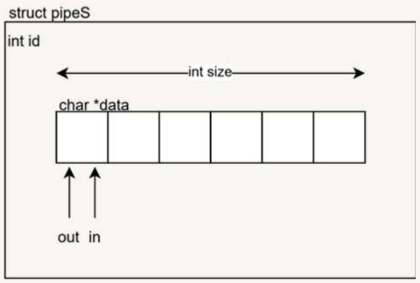
Functions

2) Pipe Write

```
function pipe_write(p, c):
    find pipe with id p in pipe_arr
    if pipe exists and is open:
        wait if pipe is full
        write c to pipe_arr[i]->data[pipe_arr[i]->in]
        increment pipe_arr[i]->in
        if pipe_arr[i]->in reaches end of array, reset to 0
        return 1 (success)
    else:
        return -1 (failure)
```

Active waiting till the other thread reads data from pipe





Active waiting till the other thread writes data or closes the pipe

3) Pipe Read

```
function pipe_read(p, c):
    find pipe with id p in pipe_arr
    if pipe exists:
        wait until pipe has data or is closed
        if pipe is empty and closed:
            destroy pipe
            return 0 (destroyed)
        else:
            read c from pipe
            increment pipe_arr[i]->out
            if pipe_arr[i]->out reaches end of array,
              reset to 0
              return 1 (success)
    else:
        return -1 (failure)
```

1.1. FIFO PIPES

Functions

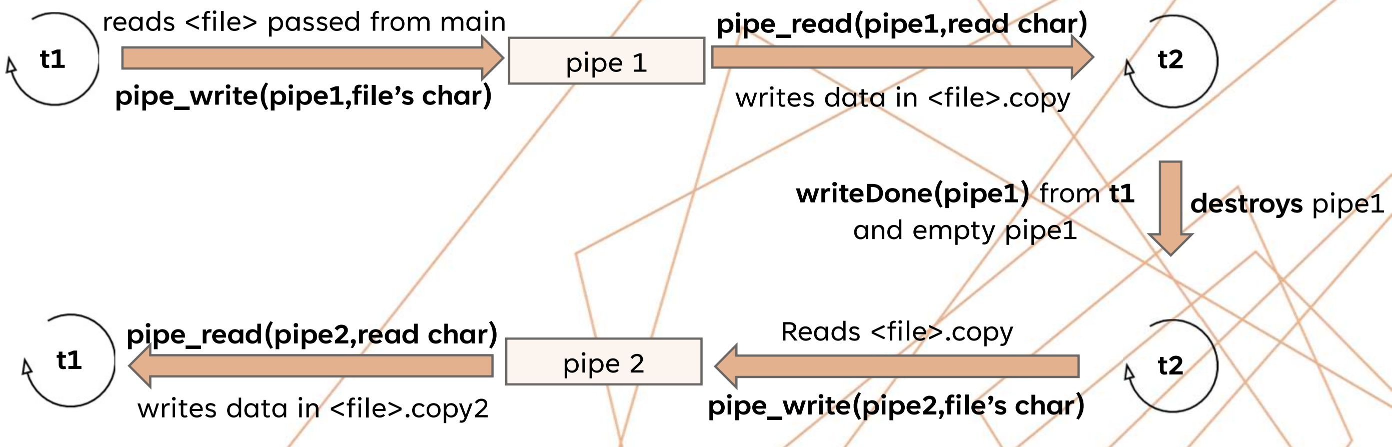
4) Pipe WriteDone

```
function pipe_writeDone(p):
    find pipe with id = p in pipe_arr
    if pipe exists and is open:
        close pipe write
        return 1 (success)
    else:
        return -1 (failure)
```

test.c

t1 : `void* thread1(void* Targs)`

t2 : `void* thread2(void* Targs)`



t1

reads <file> passed from main

pipe_write(pipe1,file's char)

pipe 1

pipe_read(pipe1,read char)

writes data in <file>.copy

t2

writeDone(pipe1) from **t1** and empty pipe1

destroys pipe1

t2

Reads <file>.copy

pipe_write(pipe2,file's char)

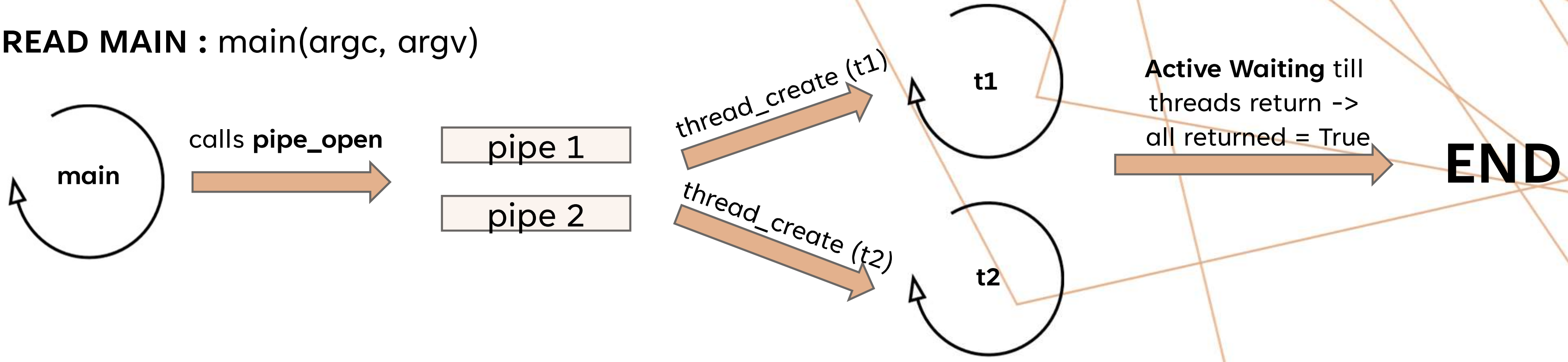
pipe 2

pipe_read(pipe2,read char)

writes data in <file>.copy2

t1

THREAD MAIN : main(argc, argv)



main

calls **pipe_open**

pipe 1

pipe 2

thread_create (t1)

thread_create (t2)

t1

t2

Active Waiting till threads return -> all returned = True

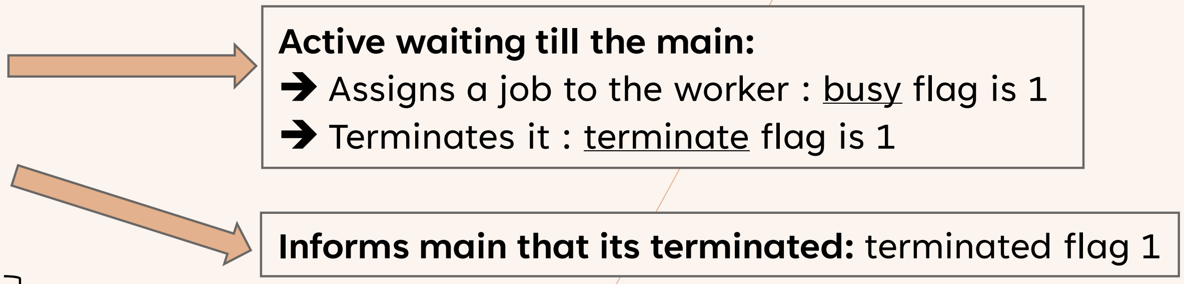
END

1.2. PRIMALITY TESTER

PrimalityTester.c

```
struct workerArgS {
    bool busy // When thread/worker is assigned a job
    int job   // Job: Int to check if its prime or not
    bool* terminate // Main variable to terminate all threads
    bool terminated // Inform main that the thread is terminated
}
```

```
void* worker(void* Wargs)

worker loop:
    while (1)
      wait until a job is assigned (busy flag is true or terminate flag is set)
        if terminate flag is set:
          print goodbye message and terminate
        else:
          get the job (number to check)
          if number is less than or equal to 1:
            print number is not prime
          else:
            use square root algorithm to check if number is prime
            print result (prime or not prime)
        reset worker state (busy flag to false)
```



Active waiting till the main:

- ➔ Assigns a job to the worker : busy flag is 1
- ➔ Terminates it : terminate flag is 1

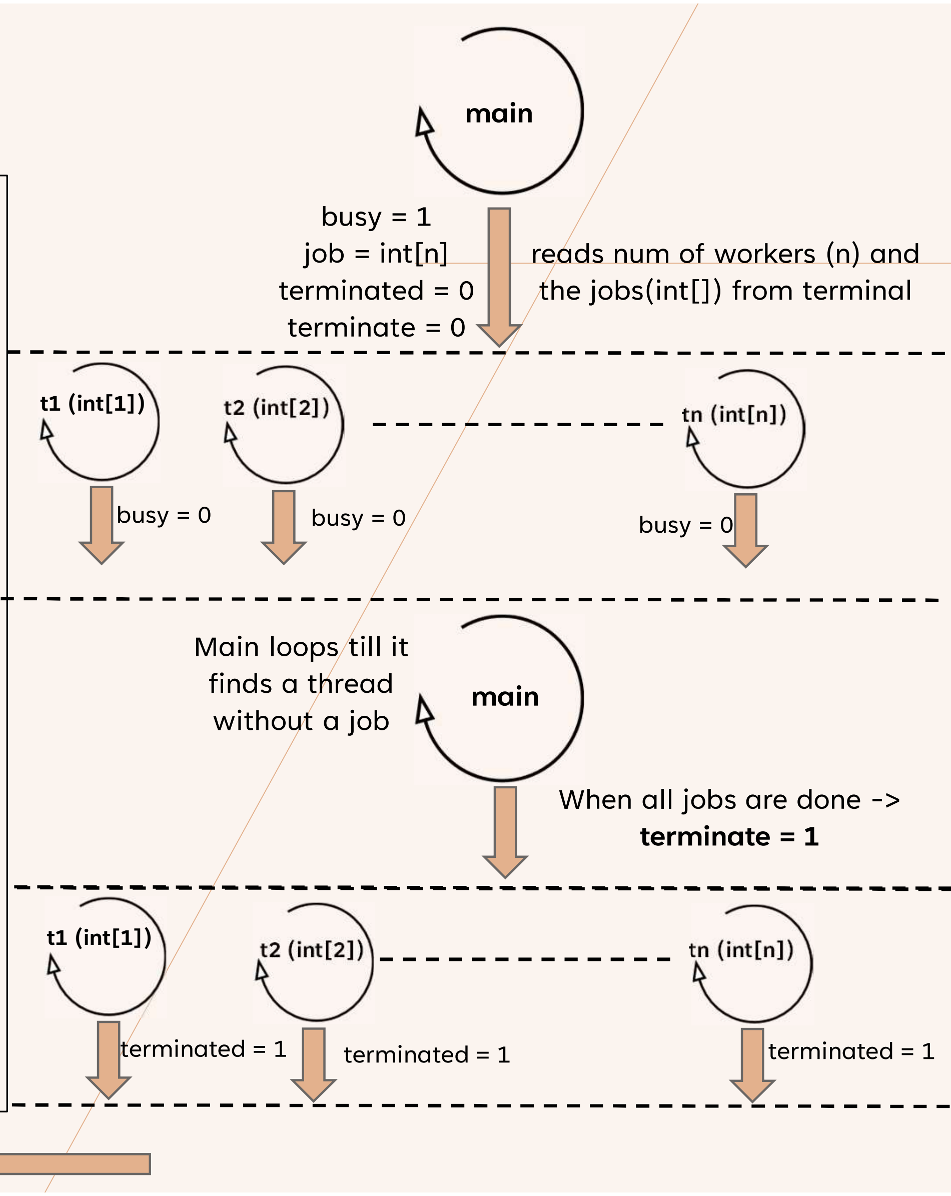
Informs main that its terminated: terminated flag 1

Geeks for geeks code.

1.2. PRIMALITY TESTER

PrimalityTester.c

```
workersNo = number of workers (from command line argument)

allocate memory for worker threads (t) and worker arguments (args)

create worker threads and initialize their arguments:
  - busy: false (not working)
  - terminate: pointer to a boolean flag (initially false)
  - terminated: flag indicating worker finished (initially false)

loop through numbers to check (from command line arguments after the first one):
  num = current number

  use round-robin approach to assign job to a worker:
    - loop until a free worker is found
    - set worker's job to num
    - set worker's busy flag to true

wait for all workers to finish their current jobs

set terminate flag to true (signal workers to stop)

wait for all workers to terminate
```



1.3. EXTERNAL MERGESORT

```
struct mergeArgS {
    char* filename // File with numbers to be sorted
    int left // Left index
    int right // Right index
    bool returned // Thread return
}
```

void* extMergeSort(void *Margs)

```
left, right, mid = get arguments (from Margs)

if (size of subarray <= 64)
  open file
  read subarray into memory
  sort subarray in memory (intMergeSort)
  reset file pointer
  write sorted subarray to file
  close file
  set returned flag to true

else (size of subarray > 64)
  initialize arguments (args1 and args2) with filename and subarray boundaries
  create two threads (t1 and t2)
  start t1 with extMergeSort and args1
  start t2 with extMergeSort and args2
  wait for both threads to finish
  merge sorted subarrays on disk (extMerge) using left, mid, and right

set returned flag to true
```

Active waiting for the threads to finish sorting the sub arrays

intMergeSort(int arr[], int left, int mid, int right)

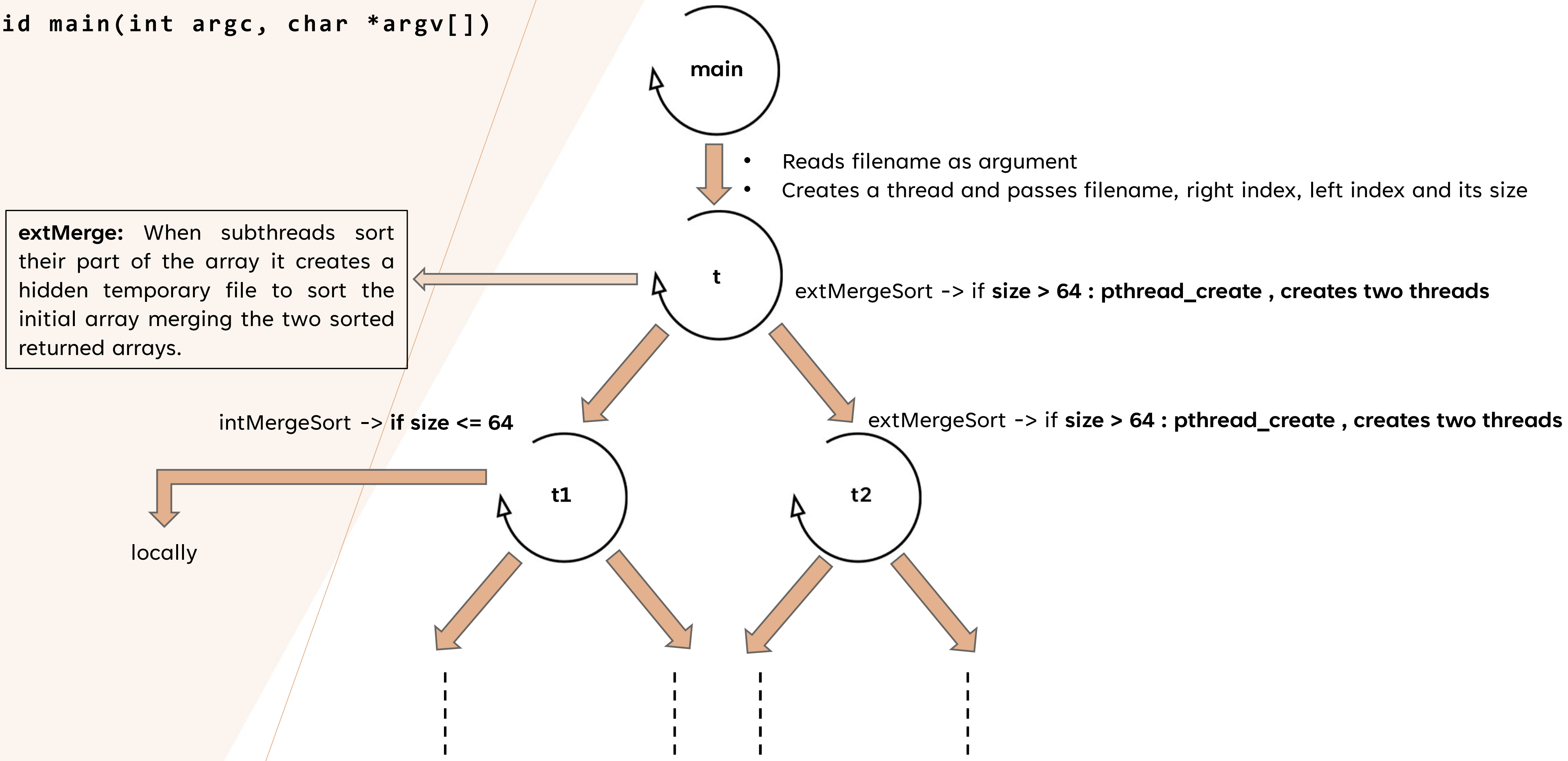
Geeks for geeks code -> Time Complexity 0(n*log(n))

void extMerge(char* filename, int left, int mid, int right)

```
size1 = left side subarray size
size2 = right side subarray size

open the file

read the first element from each subarray (leftArr and rightArr)

while (elements left in both subarrays):
  if left element is smaller:
    write left element to temporary file
    read next element from left subarray
  else:
    write right element to temporary file
    read next element from right subarray

copy remaining elements from left or right subarray (if any)

reset file pointers to the beginning

copy merged data from temporary file back to the original file
(overwriting subarrays)

close all files
remove temporary file
```

1.3. EXTERNAL MERGESORT

```
void main(int argc, char *argv[])
```



main

- Reads filename as argument
- Creates a thread and passes filename, right index, left index and its size

t

extMergeSort -> if **size > 64 : pthread_create , creates two threads**

extMerge: When subthreads sort their part of the array it creates a hidden temporary file to sort the initial array merging the two sorted returned arrays.

intMergeSort -> **if size <= 64**

locally

t1

t2

extMergeSort -> if **size > 64 : pthread_create , creates two threads**

THANK YOU